

A History of Large Language Models

■ gregorygundersen.com/blog/2025/10/01/large-language-models/

I trace an academic history of some of the core ideas behind large language models, such as distributed representations, transducers, attention, the transformer, and generative pre-training.

Published

01 October 2025

Large language models (LLMs) still feel a bit like magic to me. Of course, I understand the general machinery enough to know that they aren't, but the gap between my outdated knowledge of the field and the state-of-the-art feels especially large right now. Things are moving fast. So six months ago, I decided to close that gap just a little by digging into what I believed was one of the core primitives underpinning LLMs: the attention mechanism in neural networks.

I started by reading one of the landmark papers in the literature, which was published by Google Brain in 2017 under the catchy title *Attention is all you need* ([Vaswani et al., 2017](#)). As the title suggests, the authors did not invent the attention mechanism. Rather, they introduced a neural network architecture which in some sense “all attention”. This architecture is the now-famous *transformer*. Clearly the transformer stands in contrast to whatever came before it, but what was that and what did the transformer do differently?

To answer these questions, I read a lot of papers, and the context that felt natural to provide here grew the more that I read. I went down the rabbit hole, and when I came out, I realized that what had started as a study of attention had grown into a bigger story. Attention is still the throughline, but there are other important themes, such as how neural networks generalize and the bitter lesson that simple methods that scale seem to triumph over clever methods which do not. This post is the product of that deep dive, and it is a stylized history of LLMs.

As a caveat, real life is endlessly detailed, and any summary or synthesis inevitably flattens this detail. So I will accidentally or intentionally skip over many important and related papers and ideas in the service of a synthesis. I will also skip over practicalities such as data preprocessing and advances in hardware and computing. My focus will be on what I view as the main methodological landmarks, and this history is simply one of many ways to tell this story.

Distributed representations

I'll start with an old idea, one so ubiquitous today that it might seem silly to belabor here. The idea is that neural networks automatically generalize using *distributed representations*. This idea has its roots in computational neuroscience, particularly Connectionism ([McCulloch & Pitts, 1943](#)) and was discussed explicitly in the 1980s in papers like *Learning representations by back-propagating errors* ([Rumelhart et al., 1986](#)) and *Learning distributed representations of concepts* ([Hinton, 1986](#)). Understanding it is key to understanding why LLMs work at all and thus understanding the long line of academic research driving towards them.

But first, a problem. The goal of natural language processing (NLP) is to model human language using computers. Until the 1980s, NLP systems were mostly based on handwritten rules and handcrafted features. However, by the early 1990s, researchers were exploring the use of statistical methods from machine learning. For an early and seminal example, see *A statistical approach to machine translation* ([Brown et al., 1990](#)).

The core idea of statistical NLP is to model human language using a *statistical language model*, which is a probability distribution over all possible sequences in a language. This distribution is typically factorized such that each word depends on all words that precede it:

$$p(w_{1:T}) = \prod_{t=1}^T p(w_t | w_{1:t-1}). \quad (1) \quad p(w_{1:T}) = \prod_{t=1}^T p(w_t | w_{1:t-1}). \quad (1)$$

Throughout this post, I will use the notation $w_{i:j}$ to denote elements in a sequence from positions i to j inclusive (where $i \leq j$):

$$w_{i:j} := \{w_i, w_{i+1}, \dots, w_{j-1}, w_j\}. \quad (2) \quad w_{i:j} = \{w_i, w_{i+1}, \dots, w_{j-1}, w_j\}. \quad (2)$$

Given a good statistical model $p(w_{1:T})$, we can do many things. For example, we can rank the likelihood of different sequences of words and use that ranking to decide on things like a conversational agent's output. Or we can translate a source sequence $s_{1:T}$ into a target sequence $w_{1:T}$ if we have the conditional probabilities between the two:

$$p(w_{1:T} | s_{1:T}) \propto p(s_{1:T} | w_{1:T}) p(w_{1:T}). \quad (3) \quad p(w_{1:T} | s_{1:T}) \propto p(s_{1:T} | w_{1:T}) p(w_{1:T}). \quad (3)$$

Here, $p(w_{1:T})$ would be our *language model* of the target language, and $p(s_{1:T} | w_{1:T})$ would be our *translation model*.

Today, this view is so pervasive that it might feel obvious, but with a little imagination, I think it's easy to see how wrong this might have felt to a linguist forty-odd years ago. Equation 1 captures no language structure or parts of speech such as nouns or verbs or

adjectives—see e.g. ([Chomsky, 1956](#)) on *formal grammars*. Instead, it reduces the complexity of human language to next-word prediction. If we didn't know already that this worked, we might doubt that it would.

More importantly for us, estimating the model in Equation 11 is hard! The main challenge is the *curse of dimensionality*. There are many, many words in a vocabulary. For example, [linguists estimate](#) that English has roughly a million words, give or take a few hundred thousand depending on how you count them. Furthermore, this problem explodes in some tasks such as translation, where there are many possible conditional probabilities $p(s_{1:T} | w_{1:T})$. So when estimating the conditional probabilities of our language model, we cannot possibly encounter all possible combinations. We have a data sparsity problem, and estimating the true probabilities becomes impossible.

Perhaps the oldest idea to tackle this problem was proposed in Andrey Markov's pioneering mathematical analysis of Pushkin's *Eugene Onegin* ([Markov, 1913](#)). He made the assumption that each conditional probability in Equation 11 only depends on the previous NN terms:

$$p(w_{1:T}) = \prod_{t=1}^T p(w_t | w_{1:t-1}) \approx \prod_{t=1}^T p(w_t | w_{t-N:t-1}). \quad (4) \quad p(w_{1:T}) = \prod_{t=1}^T p(w_t | w_{t-N:t-1}). \quad (4)$$

Today, we would call this a “Markov assumption”, and Equation 44 is the famous *NN-gram model*. Particularly for small NN , say $N=1$ or $N=2$, we might be able to get reasonable estimates of data. But here is the problem, and this problem is a central theme driving towards the attention mechanism: the Markov assumption destroys *context*. Without more context, a language model can never replicate the complexity and nuance of natural language.

As I understand it, this was conceptually the state of the field circa 2000. But then in 2003, a seminal paper was published: *A neural probabilistic language model* ([Bengio et al., 2003](#)). In that paper, the authors proposed a novel idea: to avoid this data sparsity problem, this curse of dimensionality, we can use neural networks to learn a language model using what they call “distributed representations” of words. (Today, we might call these “word embeddings”.) They proposed three core ideas. First, they represented each word as a real-valued vector or embedding; then, they expressed Equation 11 in terms of these embeddings; and finally, they trained a neural network to simultaneously learn the embeddings and the parameters of the probability function (neural network) in Equation 11 using back-propagation ([Rumelhart et al., 1986](#)).

That's a lot, so let's break it down a bit. Our goal here is to learn a good model f_{Θ} of natural language such that

$$p(w_t | w_{1:t-1}) \approx f_{\Theta}(w_{t-1}, \dots, w_{t-N}). \quad (5) \quad p(w_t | w_{1:t-1}) \approx f_{\Theta}(w_{t-1}, \dots, w_{t-N}). \quad (5)$$

So the left-hand side is the true conditional distribution, capturing next-word prediction. It's the goal of language modeling. But in practice, modeling the full context is hard. So we settle for the right hand side, which is a parametric approximation f_{θ} of this true distribution with *context window* of size NN .

In Bengio, they model f_{θ} using two components. First, they represent words as vectors. Let V denote our vocabulary, which is simply a set of integers $V = \{1, 2, \dots, V\}$ indexing all V words in a language. We will represent each word as a D -vector, and so we can represent the entire language as a matrix $C \in \mathbb{R}^{V \times D}$ (Figure 11). Now for the i -th word in a sequence $w_1:T$, we have an associated index in the vocabulary, which we will denote as $I(w_i) \in V$. This notation might be a bit odd, but I'm careful here because w_i is not a well-defined mathematical object, and it cannot index C . But $I(w_i)$ is an integer and can index C , and so $c_{I(w_i)}$ is a D -dimensional vector (a row vector of C) representing the $I(w_i)$ -th word in the vocabulary, associated with the i -th word in the sequence. This vector is what we are calling an "embedding" or "distributed representation".

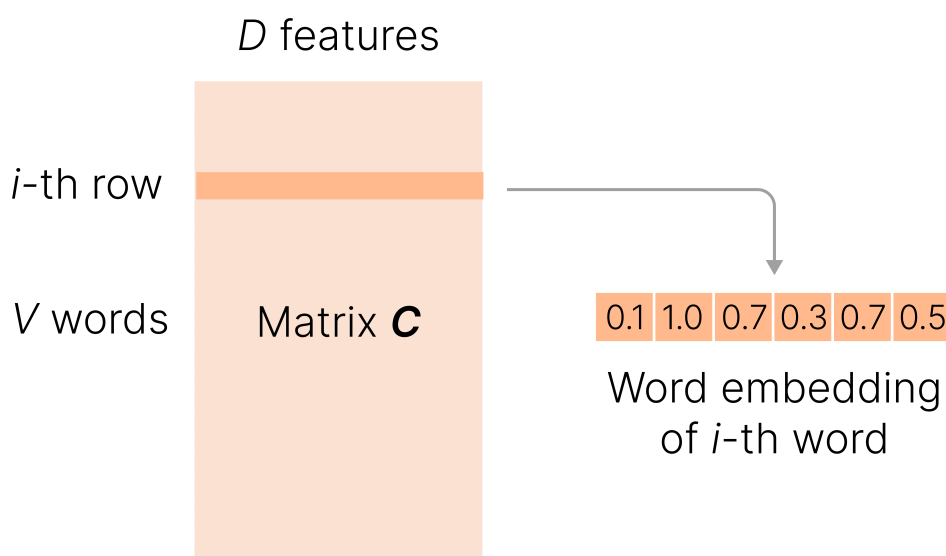


Figure 1. The matrix $C \in \mathbb{R}^{V \times D}$ represents every word in a vocabulary $V = \{1, 2, \dots, V\}$. The i -th row is a D -vector representing the i -th word, $i \in V$. This row vector is a distributed representation of that word, called a "word embedding".

Second, Bengio et al represent the probability function over words (Equation 11) as a feed-forward neural network g with parameters Ω and arguments C :

$$f_{\theta}(w_{t-1}, \dots, w_{t-N}) = g_{\Omega}(c_{I(w_{t-1})}, \dots, c_{I(w_{t-N})}). \quad (6) \quad f_{\theta}(w_{t-1}, \dots, w_{t-N}) = g_{\Omega}(c_{I(w_{t-1})}, \dots, c_{I(w_{t-N})}). \quad (6)$$

They then use back-propagation to jointly estimate the parameters

$$\theta := \{C, \Omega\}. \quad (7) \quad \Theta := \{C, \Omega\}. \quad (7)$$

In other words, they learn the neural network parameters Ω at the same time as learning the word embeddings C . Note that “distributed representation” can refer to either the continuously-valued vector, e.g. word embedding, or the concept distributed across neurons. This duality is exemplified in CC which is both a set of learnable parameters *and* the embeddings themselves!

Why might this work? The authors explain the idea so well that it’s worth just quoting the original paper:

In the proposed model, it will so generalize because “similar” words are expected to have a similar feature vector, and because the probability function is a *smooth* function of these feature values, a small change in the features will induce a small change in the probability. Therefore, the presence of only one of the above sentences in the training data will increase the probability, not only of that sentence, but also of its combinatorial number of “neighbors” in sentence space.

This is a beautiful idea. If we have word embeddings that are “well-organized” in the sense that words that play similar roles in sentences (semantically and syntactically) have similar embeddings *and* if we have a smooth function from word embeddings to probabilities, then small changes in words lead to small changes in embeddings which lead to small changes in probabilities (Figure 22).

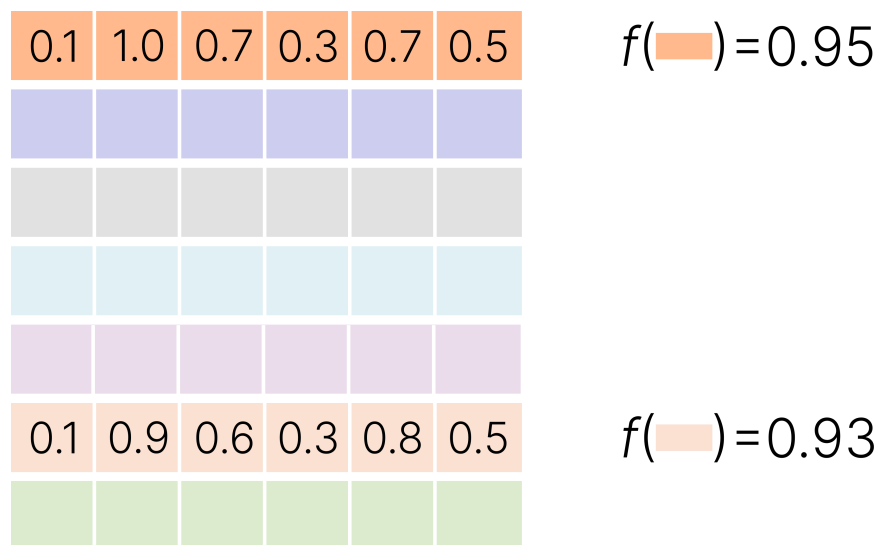


Figure 2. [\(Bengio et al., 2003\)](#) hypothesized that neural networks would generalize well on word embeddings because a small change in the input embedding could produce a small change in the output probability. Thus, if the model only saw one sentence with a given word, it might still generalize to similar sentences with similar words.

Pause for a moment to really think about this. Words are discrete objects, and a “small change in a word”, while intuitive to humans, is ill-defined. But this approach concretizes what that means. To quote the paper *Linguistic regularities in continuous space word representations* [\(Mikolov et al., 2013\)](#), which we’ll discuss later:

Whereas an NN -gram model works in terms of discrete units that have no inherent relationship to one another, a continuous space model works in terms of word vectors where similar words are likely to have similar vectors. Thus, when the model parameters are adjusted in response to a particular word or word-sequence, the improvements will carry over to occurrences of similar words and sequences.

For example, if the words “dog” and “cat” are nearby in word-embedding space, then maybe “The cat is walking on the sidewalk” and “The dog is walking on the sidewalk” should have similar probabilities. And only one of these two sentences would need to exist in the training data for the model to generalize well to both sentences!

As I mentioned, this idea was not entirely new in 2003. Since the 1980s, researchers had known that neural networks can generalize because they distribute their representation across many neurons ([Hinton, 1986](#)). Each new example modifies the weights, incorporating new knowledge into the old. However ([Bengio et al., 2003](#)) is a landmark paper in NLP because it was the first application of this idea to language modeling. The Bengio paper took seriously the idea that we could build a statistical model of language using the distributed representations of words. It was the first hint that we could use neural networks to overcome the curse of dimensionality that plagued statistical NLP.

Autoregressive framework

This is a promising idea, but we glossed over an important detail: how do we actually train this model? What is the loss function or objective that the neural network should use? And given a fit model, how do we generate a new sequence? These are important questions to answer per se, but they are also important questions because, at a conceptual level, there is really no difference between Bengio’s model and the frontier large language models today. So understanding this is critical to understanding LLMs. Both are *autoregressive* models and trained using *next-word prediction*.

As an example, imagine we have the following input sentence, which is a quote from Virginia Woolf’s *A Room of One’s Own*:

“Intellectual freedom depends upon material things.” (8)

“Intellectual freedom depends upon material things.”(8)

Now imagine that our model’s context window has size $N = 2N=2$ and let c_p denote a padding DD -vector of all zeros. In Bengio’s model, we would start by representing just the first word, “intellectual”, as a word embedding. So the first non-zero input to our model would be:

$$x_2 = \begin{bmatrix} c_p \\ c_{I(w_1)} \end{bmatrix} = \begin{bmatrix} c_p \\ c_{I(\text{“intellectual”})} \end{bmatrix}. \quad (9) \quad x_2 = [cpcl(w_1)] = [cpcl(\text{“intellectual”})]. \quad (9)$$

The output of the neural network would be a V -dimensional vector representing the probability distribution over $p(w_2 | w_1)$. Illustratively:

$$y_2 = \begin{bmatrix} p(w_2 = \text{"about"}) \\ p(w_2 = \text{"above"}) \\ \vdots \\ p(w_2 = \text{"freedom"}) \\ \vdots \\ p(w_2 = \text{"freedom"}) \end{bmatrix}. \quad (10) \quad y_2 = [p(w_2 = \text{"about"}) \ p(w_2 = \text{"above"}) \ : \ p(w_2 = \text{"freedom"}) \ : \] \cdot (10)$$

We would then compute the cross-entropy loss between this output vector and the true distribution, which is really just a one-hot vector with 1 for the word "freedom" and 0 everywhere else. We would then repeat this process on the next word. So the next input sequence would be

$$x_3 = \begin{bmatrix} c_{I(\text{"intellectual"})} \\ c_{I(\text{"freedom"})} \end{bmatrix}, \quad (11) \quad x_3 = [\text{cl}(\text{"intellectual"}) \ \text{cl}(\text{"freedom"})], \quad (11)$$

and the output would represent the probability distribution $p(w_3 | w_{1:2})$. And again, we would minimize the cross-entropy loss between its associated output vector and a one-hot vector encoding the word "depends". We would repeat this process until the end of the sentence.

Of course, longer sequences are more expensive to train in this way, and this is precisely the point of the context window in Bengio's paper. We only consider the N previous words when predicting the next word. This idea of a limited context window is critical, as it is a constraint that persists into the present day. In this example, since $N = 2$, the third input would be

$$x_4 = \begin{bmatrix} c_{I(\text{"freedom"})} \\ c_{I(\text{"depends"})} \end{bmatrix}. \quad (12) \quad x_4 = [\text{cl}(\text{"freedom"}) \ \text{cl}(\text{"depends"})]. \quad (12)$$

So the model completely loses the word "intellectual". It is now outside the context.

Since minimizing the cross-entropy loss is equivalent to maximizing the log likelihood—see [here](#) for an example if this idea is new to you—we can generalize the logic above by saying that we want to maximize the log likelihood of our training data, again using a neural network as a parametric function approximation of the true distribution:

$$\theta^* = \arg \max_{\theta} \left\{ \sum_{t=1}^T \log g_{\Omega} (c_{I(w_{t-N})}, \dots, c_{I(w_{t-1})}) \right\}. \quad (13)$$

$$\Theta^* = \arg \max_{\Theta} \left\{ \sum_{t=1}^T \log g_{\Omega} (c_{I(w_{t-N})}, \dots, c_{I(w_{t-1})}) \right\}. \quad (13)$$

Of course, we can estimate θ^* by minimizing the negative log likelihood using gradient descent via back-propagation. That's it. At the conceptual level, this framework is no different from how frontier large language models are trained today. As we will see later though, there is a lot of additional machinery that is needed to make these models work in practice.

Finally, imagine we fit our model, meaning we find good parameters θ^* that maximize our log likelihood. How can we use these parameters to generate a random sequence or sentence? We could draw the first word at random from the vocabulary. And then we could draw the next word conditional on the first word from our parametric approximation of $p(w_2 | w_1)$. And then we could draw the third word conditional on the second and first words from our parametric approximation of $p(w_3 | w_{1:2})$. And so on. This is why LLMs can both understand natural language and generate new sentences. They are not just descriptive models; they are [generative models](#).

There are some subtleties I am glossing over, such as special embeddings to denote the start and end of a sequence, preprocessing steps like lowercasing words, tokenization, and handling out-of-vocabulary words. But I don't think these details matter much here.

As an aside, we can call any model trained in this way *autoregressive*. In statistics, an autoregressive model is any model where a variable is predicted using its own previous values. A classic example of this are [AR models](#) such as AR(1).

The breakthrough

While [\(Bengio et al., 2003\)](#) was a landmark paper, its full impact was delayed by roughly a decade. This is because training neural networks was hard at the time. It's worth checking out that paper and seeing just how primitive the engineering feels today. For example, they trained on CPUs and without modern tooling like automatic differentiation libraries.

In the intervening decade, there was some early work that built on Bengio's model. For example, in *A unified architecture for natural language processing: Deep neural networks with multitask learning* [\(Collobert & Weston, 2008\)](#), the authors demonstrate that Bengio's neural language model could be trained and used on a variety of downstream tasks. And in *Word representations: A simple and general method for semi-supervised learning* [\(Turian et al., 2010\)](#), the authors demonstrate that word embeddings improve state-of-the-art NLP systems when included as additional features. But none of these contributions were convincing demonstrations of Bengio's main idea.

So seven years after Bengio et al, it was N N-grams, not neural networks, which were still the state-of-the-art, at least in practice and outside specialized benchmarks. Honestly, I found this surprising, but I kept reading this claim in various papers. For example, in the introduction to *Recurrent neural network based language model* ([Mikolov et al., 2010](#)), the authors wrote:

It is questionable if there has been any significant progress in language modeling over simple N N-gram models... In fact, most of the proposed advanced language modeling techniques provide only tiny improvements over simple baselines, and are rarely used in practice.

Or two years after that, in *A fast and simple algorithm for training neural probabilistic language models* ([Mnih & Teh, 2012](#)), the authors wrote:

In spite of their superior performance, neural probabilistic language models remain far less widely used than N N-gram models due to their notoriously long training times, which are measured in weeks even for moderately-sized datasets.

Of course, advanced techniques existed and were well known, but they were often impractical. So roughly a hundred years after Andrey Markov's pioneering work, researchers were still struggling to represent human language in a form amenable for mathematics and computation, and N N-grams were still considered a reasonable choice in NLP. Today, neural networks are definitively state-of-the-art. What changed? The answer is that we learned to train variants of Bengio's model at scale.

Around 2012, researchers were finally able to train neural networks on large datasets. My understanding is that it was the so-called "AlexNet" paper, *ImageNet classification with deep convolutional neural networks* ([Krizhevsky et al., 2012](#)), that convinced many in the research community to pay attention. Convolutional neural networks were already well known and had been trained on small datasets since the 1980s ([LeCun et al., 1989](#)). But AlexNet was the first time a deep convolutional neural network was trained end-to-end on a very large (at the time) dataset, ImageNet ([Deng et al., 2009](#)) and using GPUs. The results were a tour de force. To quote the paper:

We also entered a variant of this model in the ILSVRC-2012 competition and achieved a winning top-5 test error rate of 15.3%15.3%, compared to 26.2%26.2% achieved by the second-best entry.

In other words, AlexNet demolished the state-of-the-art in computer vision. It achieved a roughly 40%40% reduction in relative error rate. Nothing else came close. As a comparison, the current fastest time for a men's marathon is 2 hours and 35 seconds. The previous record was 2 hours and 69 seconds, so 34 seconds slower. Now imagine if someone came along and beat the record by half an hour. It would revolutionize the running world.

At the time, computer vision was still dominated by handcrafted feature pipelines, and so the AlexNet results were extremely surprising. For example, in *Introduction to the bag of features paradigm for image classification and retrieval* ([O'Hara & Draper, 2011](#)), the authors wrote:

The past decade has seen the growing popularity of Bag of Features (BoF) approaches to many computer vision tasks, including image classification, video search, robot localization, and texture recognition... BoF-based systems have set new performance standards on popular image classification benchmarks and have achieved scalability breakthroughs in image retrieval.

This introduction to bag of feature models was put on arXiv in January 2011, whereas AlexNet was published at NeurIPS in December 2012, meaning that the claim above was contemporaneous with the training of AlexNet! My point here is to underscore just how surprising the rise of neural networks was. To be clear, I am sure many in the research community believed neural networks would work—Hinton has been a believer since probably the 1970s—, but this was hardly the consensus view that it is today. So the year 2012 was a changepoint. In 2003, Bengio et al set the stage conceptually. In 2012, Krizhevsky et al set the stage technologically.

Shallow log-linear models

With hindsight, the obvious implication of AlexNet was that NLP researchers circa 2012 should try to train neural networks at scale. Of course, many researchers tried, but let's ground ourselves in one particular model. This will help focus the narrative.

To my knowledge, two of the earliest and most successful papers to try this idea were *Efficient estimation of word representations in vector space* ([Mikolov et al., 2013](#)) and *Distributed representations of words and phrases and their compositionality* ([Mikolov et al., 2013](#)). These papers are tightly related by both authorship and time, and together, they helped unlock the core ideas in Bengio's paper, as well as introduce the famous [word2vec](#) model. So I think it's fair to treat them as both a unit and as a landmark in our story.

To understand these two papers, we need to understand the computational problems Bengio faced, which means we need to understand the model in more technical detail. Let x_t be the input to the model, and y_t be the output. Bengio's model did not support variable-length inputs, and thus the input sequence could be only a fixed number of N words, each represented as an D -dimensional embedding. Let's represent this input as the concatenation of N different D -vectors from C mentioned above, so:

$$x_t := \begin{bmatrix} c_{I(w_{t-1})} \\ \vdots \\ c_{I(w_{t-N+1})} \end{bmatrix}. \quad (14) \quad x_t = [c(w_{t-1}) : c(w_{t-N+1})]. \quad (14)$$

One way we can imagine constructing x_t is if we represent every word in our context window as a V -dimensional one-hot vector. Call this a matrix $Q_t \in R^{N \times V}$. Then $x_t = Q_t C$ gives us the associated embeddings. In practice, though, we would never do a dense matrix multiplication with complexity $O(VND)$. Instead, we would simply index into C . So this operation has computational complexity $O(ND)$. I only belabor this point because I found it confusing when first reading Bengio's paper. (This point is made more clearly in [\(Collobert & Weston, 2008\)](#))

After construction, this input x_t is then fed into an extremely simple (relative to today's models) architecture, a feed-forward neural network with a linear projection layer and a nonlinear hidden layer:

$$\begin{aligned} g_{\Omega}(x_t) &= y_t & : &= b + Wx_t + U \tanh(z_t), \\ z_t & & : &= d + Hx_t. \end{aligned} \quad (15) \quad g_{\Omega}(x_t) = y_t z_t = b + Wx_t + U \tanh(z_t) \\), := d + Hx_t. (15)$$

The output $y_t \in R^V$ represents the un-normalized probability of each word in the vocabulary. If normalized, this vector would represent the probability distribution we discussed in the autoregressive framework.

Here, we see that that $W \in R^{V \times ND}$ $W \in R^{V \times ND}$ is a linear projection of the input embeddings x_t to ND , that $H \in R^{H \times ND}$ $H \in R^{H \times ND}$ is a linear projection into a hidden state vector $z_t \in R^H$ $z_t \in R^H$, and that $U \in R^{V \times H}$ $U \in R^{V \times H}$ is a linear projection of the nonlinear hidden state vector. So clearly the parameters mentioned in Equation 77 can be concretized as

$$\{C, \Omega\} := \{C, b, W, U, d, H\}. \quad (16) \quad \{C, \Omega\} := \{C, b, W, U, d, H\}. \quad (16)$$

So why was this expensive to train? We can see that the computational complexity to compute y_t is proportional to:

$$\underbrace{ND}_{QC} + \underbrace{VND}_{Wx_t} + \underbrace{VH}_{U \tanh(z_t)} + \underbrace{HND}_{Hx_t}. \quad (17) \quad \text{QC}$$

Note that this complexity is for every single word in the corpus, and we must also account for the number of training epochs. In (Mikolov et al., 2013), the authors write that a “common choice” is $N = 10$ and that ND is typically around 500 to 2000. However, the hidden layer has dimension H (commonly around 2000 or so) and this

is multiplied by the size of the vocabulary! What does this mean? The dominating term in Equation 1717 is $VHVH$. Furthermore, this complexity is just for computing the un-normalized probabilities $y_t y_t$. To normalize these, we must compute the softmax function over the size of the vocabulary VV :

$$p(w_t | w_{t-N:t-1}) = \frac{\exp(y_t)}{\sum_{i=1}^V \exp(y_i)}. \quad (18) \quad p(w_t | w_{t-N:t-1}) = \frac{\exp(y_t)}{\sum_{i=1}^V \exp(y_i)}. \quad (18)$$

As I understand it, these were the computational problems Bengio faced. The two Mikolov papers did not present a single trick to solve them. Rather, the papers made a number of modeling choices, mostly already established in the literature, that in combination finally made learning distributed representations of words scalable.

First, in the first paper, they avoided computing the full softmax function using hierarchical softmax, introduced by Morin and Bengio in *Hierarchical probabilistic neural network language model* ([Morin & Bengio, 2005](#)). I don't think the details of this matter much here. See [this blog post](#) for a nice explanation with code. Suffice to say that it's an efficient way to compute the normalized probabilities in Equation 1818. The computational complexity is reduced from $O(V)$ to $O(\log V)$. In the second paper, they further

sped up the softmax computation by introducing a technique called *negative sampling*. The theory here is rich and deserving of its own post, but the main idea is to draw K samples from a noise distribution and train the model to disambiguate observations from noise. The important point here is that one can prove this converges to the correct probabilities without explicitly computing the normalizing constant. See ([Gutmann & Hyvärinen, 2010](#)) for details. We don't need to fully grok these techniques; just know that these two approaches are both ways of getting around the expensive normalization in Equation 1818. For example, if $V = 1 \times 10^6$, then $\log(V) \approx 20$. And

in the second paper, they chose K to be 22 to 2020 depending on the dataset.

Second, they stripped out the non-linear part of Bengio's model (so removing $\tanh(z_t)$), reducing the model to a simple linear operation: a dot product. The result is model that is log-linear on the features, which I'll explain in a moment.

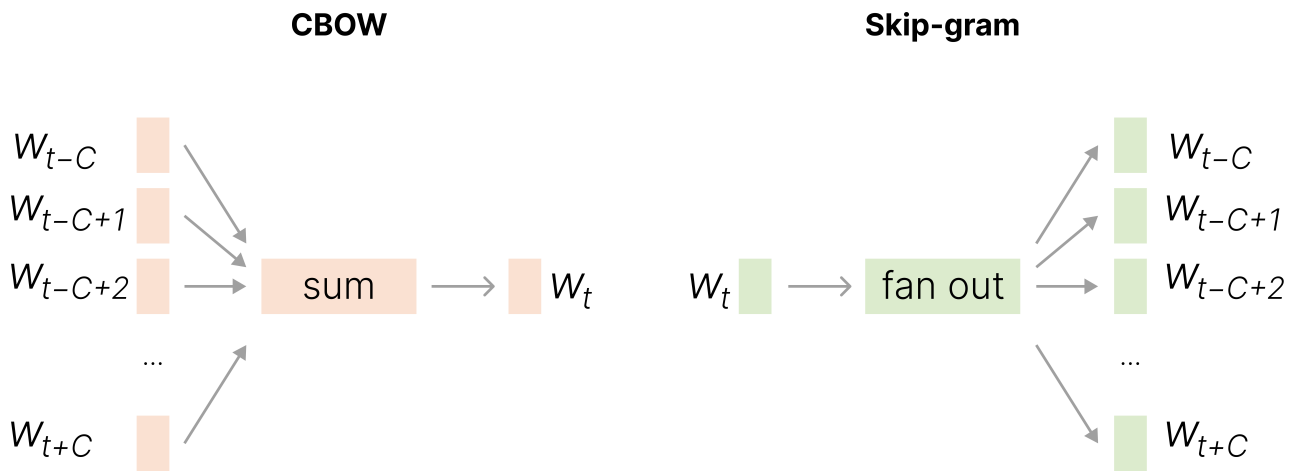


Figure 3. Diagrams of the continuous bag-of-words model or CBOW (left) and the skip-gram model (right). CBOW is trained such that neighboring words predict a target word. Skip-gram is trained such that a center word predicts its neighboring words. Both are simple log-linear models.

Now the models. In the first paper, they presented two models, a continuous bag-of-words model (CBOW) and a continuous skip-gram model (skip-gram). These are the foundations of the [word2vec](#) NLP toolkit. In the CBOW model, a set of neighboring words are averaged to predict a target word; and in the skip-gram model, a target word is used to predict its neighboring words (Figure 33). Both worked empirically in practice, but the authors only built on the skip-gram model in the second paper. And since I don't think it's that important here to understand both, I'll just focus on the skip-gram model.

Let's build a little intuition by going into detail. The objective of the skip-gram model is to minimize the cross-entropy loss between a single target word and its neighboring words. So the input to the model is only a single DD -vector representing a single word (so no context window). The output, however, are the NN words surrounding the input. Let $N = 2CN = 2C$. Then the objective function is:

$$\frac{1}{T} \sum_{t=1}^T \sum_{-C \leq j \leq C, j \neq 0} \log p(w_{t+j} | w_t). \quad (19) \quad T1t=1 \sum_{t=1}^T \sum_{-C \leq j \leq C, j \neq 0} \log p(w_{t+j} | w_t). \quad (19)$$

I will continue to use the notation NN for this context window, but clearly it is different in precise meaning from the NN in an NN -gram or the NN in Bengio's paper. We model the conditional probability in Equation 19 via a simple log-linear function:

$$p(w_{t+j} | w_t) = p(u_{I(w_{t+j})} | c_{I(w_t)}) = \frac{\exp(\langle u_{I(w_{t+j})}, c_{I(w_t)} \rangle)}{\sum_{i \in V} \exp(\langle u_i, c_{I(w_t)} \rangle)} \quad (20) \quad p(w_{t+j} | w_t) = p(u_{I(w_{t+j})} | c_{I(w_t)}) = \sum_{i \in V} \exp(\langle u_i, c_{I(w_t)} \rangle) \exp(\langle u_{I(w_{t+j})}, c_{I(w_t)} \rangle) \quad (20)$$

Here, c_i are word embeddings of the inputs. These are analogous to the row-vectors of CC in Bengio's model and again are constructed via a lookup. The output embeddings u_i are a little trickier to interpret. If we were using the full softmax function, we would have V

V such output embeddings, and these would represent the weights of the softmax function. But when using hierarchical softmax or negative sampling, the interpretation changes a bit. Again, I don't think the details really matter here. The key point is that we take a sequence $w_1:T$, select the appropriate embeddings $c_1:T$, and compute Equation 20 directly, learning both the parameters CC and UU .

This is called a "log-linear model" because the log of the conditional probability is linear with respect to its arguments:

$$\log p(w_{t+j} | w_t) = \langle u_{I(w_{t+j})}, c_{I(w_t)} \rangle - Z, \quad (21) \quad \log p(w_{t+j} | w_t) = \langle u_{I(w_{t+j})}, c_{I(w_t)} \rangle - Z,$$

(21)

Here, I just write Z to denote the normalizing constant, the denominator in Equation 20, because it is not particularly interesting, and we do not even need to compute it when using negative sampling. The key relationship that the model is learning is a simple linear weighting of the input embeddings that allow it to predict nearby words.

Hopefully, it is clear why this model is so fast to train. We have no hidden layers or nonlinearities. We simply compute a dot product and ignore the normalizing constant. For example, when using the full softmax, the computational complexity is:

$$N(D + DV). \quad (22) \quad N(D+DV).$$

Here, we have $D + DV$ dot products, and we need to do it over N words in our context window. However, in practice, we can eliminate V entirely, replacing it with something around $\log(V)$ or $\log_2(V)$. This is significantly smaller than Equation 17

2
17. For example, if we assume that $H = D = 500$, $N = 10$, and $V = 1 \times 10^6$, then hierarchical softmax is five orders of magnitude smaller in terms of complexity.

So in these two seminal Mikolov papers, the authors stripped down Bengio's core idea to a simple log-linear model, and thus were able to train that model at scale. That said, I want to stress a subtlety that took me time to grok. Neither the CBOW nor the continuous skip-gram models presented here are full language models. Notice that their objective functions (nearby-word prediction) are not in the autoregressive framework and thus cannot easily plug into Equation 11. That's because the goal of these papers was not to learn a full language model but rather to learn good word embeddings. They say this explicitly in the first paper (emphasis mine):

Representation of words as continuous vectors has a long history. A very popular model architecture for estimating neural network language model (NNLM) was proposed in ([Bengio et al., 2003](#)), where a feed-forward neural network with a linear projection layer and a non-linear hidden layer was used to learn jointly the word vector representation and a statistical language model. This work has been followed by many others.

Another interesting architecture of NNLM was presented in ([Mikolov, 2007](#); [Mikolov et al., 2009](#)), where the word vectors are first learned using neural network with a single hidden layer. The word vectors are then used to train the NNLM. Thus, the word vectors are learned even without constructing the full NNLM. *In this work, we directly extend this architecture, and focus just on the first step where the word vectors are learned using a simple model.*

So the word2vec models were simple and shallow (single layer) neural networks designed for fast training and to learn good embeddings. They were not full language models. This is a major distinction from similar prior art, such as *A scalable hierarchical distributed language model* ([Mnih & Hinton, 2008](#)). In this paper, the authors demonstrate more scalable inference of Bengio's model by representing the vocabulary compactly through binary trees and by using a log-bilinear model. But they go end-to-end to a language model, as the paper title suggests. Mikolov et al's two models were relentlessly simple and efficient.

As I understand it, both CBOW and skip-gram worked well in practice. It did not matter if neighboring words predict a target word or if that target word predicts its neighboring words. The real differentiator was that both models could be efficiently trained at scale. And with scale, something remarkable happened: the authors discovered that distributed representations of words, trained in this fashion, captured semantic and syntactic information.

Emergent linguistic regularities

Today, linguistic regularities in word embeddings is so well-established that it might seem boring to read here. But understood in context, these regularities should be surprising! How can a simple linear model, trained on essentially next- or nearby-word prediction via maximum likelihood estimation, learn distributed representations of words with remarkable syntactic and semantic properties and relationships? In my mind, this was the first big result that suggested neural networks would not just work but *really work* in language modeling.

The word2vec papers were not the first to observe these properties. My understanding is that that credit goes to yet another Mikolov paper from 2013, *Linguistic regularities in continuous space word representations* ([Mikolov et al., 2013](#)). Here, the authors showed that many semantic and syntactic relationships correspond to approximately constant vector offsets in the embedding's vector space.

To be clear, researchers had long observed that one could uncover structure in vector representations of words. For example, in the 1989 paper *Self-organizing semantic maps* ([Ritter & Kohonen, 1989](#)), the authors trained self-organizing maps ([Kohonen, 1982](#)) on pre-computed two-dimensional vectors representing words and demonstrated that these maps contain semantic structure. However, these models were not trained end-to-end (the representations themselves were not learned) and did not have linear structure. It would be a stretch to call these vectors “word embeddings”. But log-linear models like word2vec were remarkable precisely because they enabled analogical reasoning through simple vector offset, i.e. linear operations (Figure 44)!

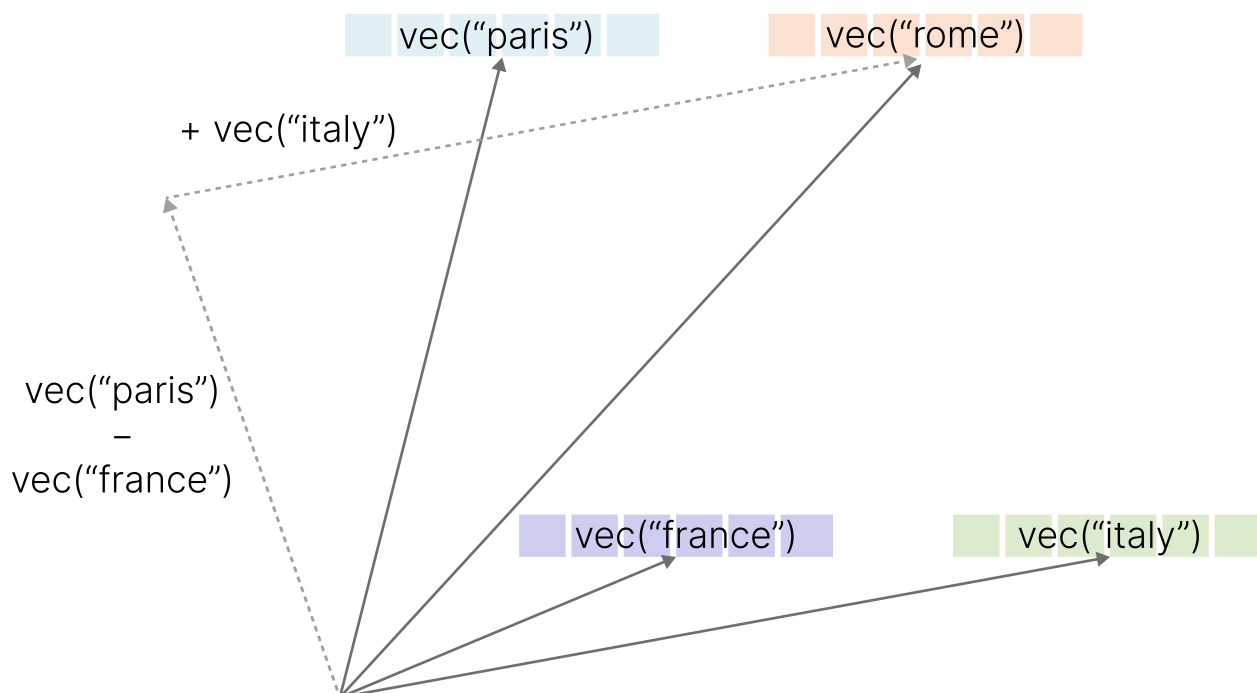


Figure 4. Word embeddings can capture semantic and syntactic regularities in natural language. Here, the vectors for "paris", "rome", "france", and "italy" are illustrated. The vector algebra of "paris - france + italy" gives us a vector quite close to "rome". This suggests that word embeddings are embedded in a vector space that captures some of the regularities of natural language.

Perhaps the most famous example of analogical reasoning with word embeddings is the relationship “king is to queen as man is to woman”:

$$\text{vec}(\text{"king"}) - \text{vec}(\text{"man"}) + \text{vec}(\text{"woman"}) \approx \text{vec}(\text{"queen"}). \quad (23)$$

$$-\text{vec}(\text{"man"}) + \text{vec}(\text{"woman"}) \approx \text{vec}(\text{"queen"}) - \text{vec}(\text{"king"}). \quad (23)$$

Or in ([Mikolov et al., 2013](#)), the authors give the example that “Russia” plus “river” is the Volga:

$$\text{vec}(\text{"Russia"}) + \text{vec}(\text{"river"}) \approx \text{vec}(\text{"Volga River"}). \quad (24)$$

$$\text{vec}(\text{"Russia"}) + \text{vec}(\text{"river"}) \approx \text{vec}(\text{"Volga River"}). \quad (24)$$

In my mind, these are pretty fascinating and non-obvious results. It suggests that the methods are not mixing vector dimensions in undesirable ways and staying approximately linear. Again, viewed with fresh eyes, it is really quite remarkable! If you were a researcher in 2003 reading Bengio's paper, would you have predicted this result with high confidence?

While these two Mikolov papers are landmark papers on learning word embeddings at scale, they are by no means the only ones. Many other researchers worked in this area. Perhaps the most famous paper on word embeddings that we do not have time to discuss is *GloVe: Global vectors for word representation* ([Pennington et al., 2014](#)). In this paper, the authors present a unifying view between two common methods for learning word embeddings, global matrix factorization methods and local context window methods. But there were many others as well, such as *Skip-thought vectors* ([Kiros et al., 2015](#)), *Word embeddings through Hellinger PCA* ([Lebret & Collobert, 2013](#)), and *Eigenwords: spectral word embeddings* ([Dhillon et al., 2015](#)) to cite just a few illustrative examples.

For ease of presentation, I have focused on word-level embeddings. But the idea was naturally and quickly extended to larger contexts. This was motivated by the fact that a word's meaning is obviously context-dependent (polysemy). For example, the word "bank" might refer to a financial institution or the side of a river. A word embedding for "bank" that is not context dependent must somehow flatten this distinction. So lack of context is obviously a limitation.

Researchers tackled this through a variety of approaches. One approach was to use the hidden states of a bidirectional long short-term memory network (LSTM) as context-specific embeddings as in *context2vec: Learning generic context embedding with bidirectional LSTM* ([Melamud et al., 2016](#)) or *Learned in translation: contextualized word vectors* ([McCann et al., 2017](#)). But perhaps the most noteworthy example of this idea—and one I mention here because it will come up later—was *Deep contextualized word representations* ([Peters et al., 2018](#)) or ELMO. Here, the authors both used a bidirectional LSTM to extract more context-dependent word embeddings and then trained on an objective function that was dependent on the downstream task. This hints at combining pre-trained embeddings with supervised fine-tuning, which we'll see later.

Long-range dependencies

By 2013, word- and phrase-level embeddings demonstrably worked. The key to unlocking them was simple methods that scaled on modern hardware. However, the problem with these embeddings is that they were still with respect to a fixed window. It was not immediately obvious how this idea could be extended to longer phrases or sentences or to larger texts. Of course, researchers had tried. For example, ([Collobert & Weston, 2008](#)) used the idea of time-delay neural networks ([Waibel et al., 1989](#)) to model sentences of variable lengths, but the authors used convolutions that still had a fixed-width window size. The embedding itself, then, was not constructed while accounting for long-range

dependencies. So word embeddings, while a beautiful idea, only set the stage for the next big idea in our history: tackling the problem of modeling long-range dependencies without an explicit context window.

The key innovation here was *sequence-to-sequence* models. In a sequence-to-sequence model, a neural network encodes a variable-length input sequence into a fixed-length vector, while a second neural network decodes this fixed-length vector back into a variable-length output sequence. In both Bengio and Mikolov's papers, the input was an embedding (cc in Equations 1414 and 2020). In a sequence-to-sequence model, this intermediate fixed-length vector is now the word embedding. The precise architectures used for the encoder and decoder can vary, but clearly they should be architectures that support variable-length sequences, such as recurrent neural networks (RNNs) or LSTMs.

To me, the most intuitive example of a sequence-to-sequence model is a translation model. The input sequence is a sentence in a *source language* like English, and the output sequence is a sentence in a *target language* like Chinese (Figure 55). And since some of the most important early work in sequence-to-sequence modeling was in *neural machine translation* (NMT), I'll often use translation as a default example. However, the more general case is any mapping from one sequence to another.

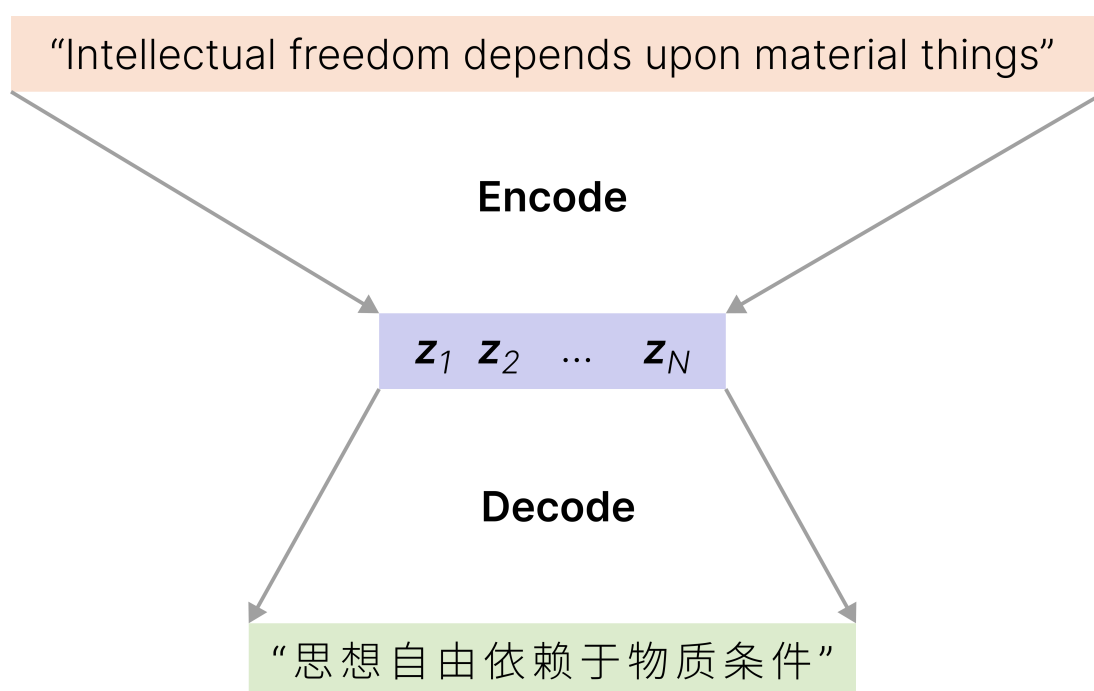


Figure 5. An illustrative example of a sequence-to-sequence model, a neural machine translator. The input sequence is a sentence in English and the output sequence is the same sentence, translated into Chinese. The information in the variable-length input is compressed into hidden variables, which are then used to generate the variable-length output. NB: I translated this sentence with an LLM; apologies for any mistranslation.

This idea is fairly straightforward; it is analogous to an auto-encoder but for variable-length sequences, and auto-encoders ([Bourlard & Kamp, 1988](#)) are nearly as old as back-propagation. However, as we have already seen, even seemingly simple ideas are

hard-won. The original work in RNNs and LSTMs goes back to at least the early 1990s, with seminal papers like *Finding structure in time* ([Elman, 1990](#)), *Serial order: A parallel distributed processing approach* ([Jordan, 1997](#)) and *Long short-term memory* ([Hochreiter & Schmidhuber, 1997](#)). By the 2010s, these sequential models were well-known and already used in NLP. See ([Mikolov et al., 2010](#); [Sutskever et al., 2011](#); [Graves, 2013](#)) for example. These models were an important bridge, proving that we could train RNNs at scale and overcome the vanishing gradient problem discussed in *Learning long-term dependencies with gradient descent is difficult* ([Bengio et al., 1994](#)). But they were not yet sequence-to-sequence models.

To my knowledge, the first paper to propose a full encoder–decoder architecture for NLP was *Recurrent continuous translation models* ([Kalchbrenner & Blunsom, 2013](#)). Here, the authors proposed training two neural networks end-to-end. The decoder was an RNN, inspired by the model in ([Mikolov et al., 2010](#)). But somewhat surprisingly, the encoder was not also an RNN. With hindsight, two RNNs feels like the obvious choice, but instead the authors used a *convolutional sentence model* (CSM). The details don't really matter here, but this is essentially an NLP model which uses convolutional layers. Why this choice? Well, CSMs were actually developed by the same authors in the same year, in *Recurrent convolutional neural networks for discourse compositionality* ([Kalchbrenner & Blunsom, 2013](#)), and my hypothesis is that this choice just felt obvious to them at the time.

So ([Kalchbrenner & Blunsom, 2013](#)) was a landmark paper in the sense that it was the first attempt at a sequence-to-sequence model, but with hindsight we can immediately see how to improve it with a better sequential model for the encoder. And that is precisely what happens in two follow up papers. First, in *Learning phrase representations using RNN encoder–decoder for statistical machine translation* ([Cho et al., 2014](#)), the authors propose the first encoder–decoder architecture in which both neural networks were RNNs. And then in *Sequence to sequence learning with neural networks* ([Sutskever et al., 2014](#)), the authors proposed a similar model but using LSTMs, since LSTMs often work better at handling the aforementioned vanishing gradient problem. In this paper, Sutskever makes the connection to Kalchbrenner explicitly:

Our work is closely related to Kalchbrenner and Blunsom, who were the first to map the input sentence into a vector and then back to a sentence, although they map sentences to vectors using convolutional neural networks, which lose the ordering of the words.

As a nitpick, convolutional neural networks do model local patterns and order, but they lose global order without very large [receptive fields](#). But Sutskever's point is directionally correct. So even at the time, the academic history we are tracing here was clear.

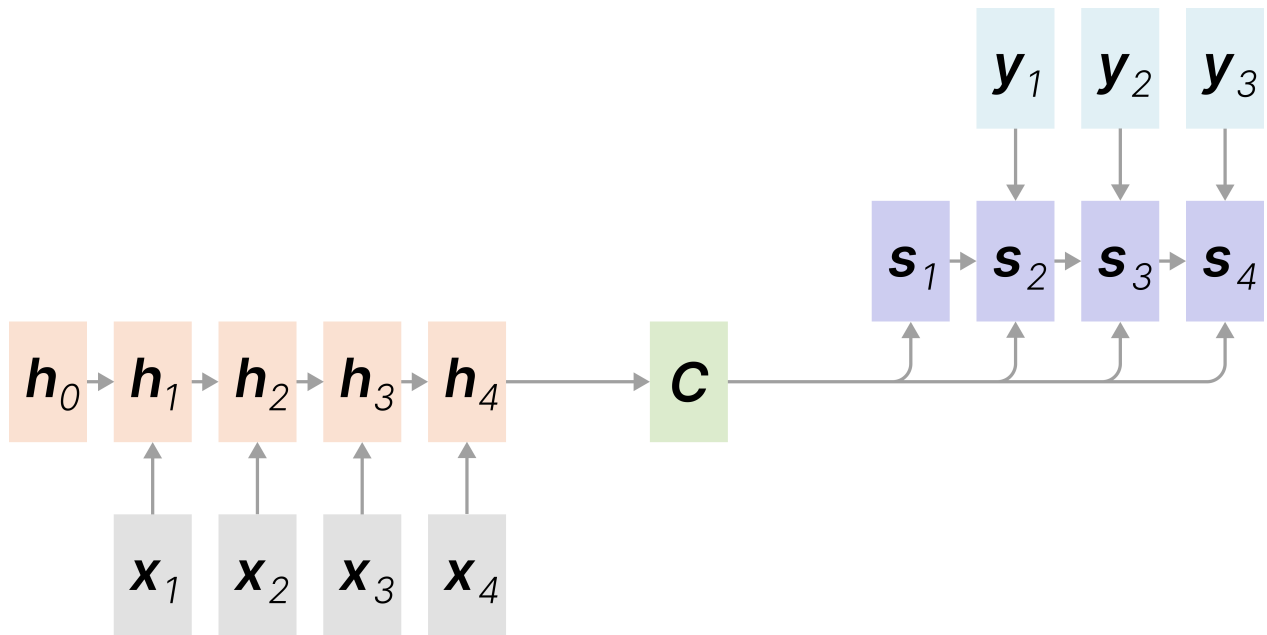


Figure 6. Diagram of the relationship between the encoder hidden states h , the decoder hidden states s , and the context vector c in the RNN encoder–decoder proposed by (Cho et al., 2014). Any sequential structure modeled by the encoder is ultimately flattened into the fixed-width context vector c .

To understand these models in a bit more detail, let's go through the RNN encoder–decoder in (Cho et al., 2014), using Figure 66 as a reference. Let X be a variable-length input sequence with length T_x , and let Y be a variable-length output sequence with length T_y :

$$\begin{aligned} X &= \{x_1, x_2, \dots, x_{T_x}\}, \\ Y &= \{y_1, y_2, \dots, y_{T_y}\}. \end{aligned} \quad (25) \quad XY = \{x_1, x_2, \dots, x_{T_x}\}, \{y_1, y_2, \dots, y_{T_y}\}. \quad (25)$$

Note that (X, Y) is a single observation pair, but I am suppressing the sample index for ease of notation. Also, I bold each vector in both sequences because they are embedded words.

In an RNN, we iteratively compute hidden state variables over T_x steps, where for the t -th step we define a recurrence relation between hidden states as:

$$h_t = f_{\text{enc}}(h_{t-1}, x_t). \quad (26) \quad h_t = f_{\text{enc}}(h_{t-1}, x_t). \quad (26)$$

This might be a little abstract. So concretely, a simple RNN network might instantiate f_{enc} as the following nonlinear function of the current word embedding and the previous hidden state:

$$h_t = \tanh(W_{hh}h_{t-1} + W_{hx}x_t). \quad (27) \quad h_t = \tanh(W_{hh}h_{t-1} + W_{hx}x_t). \quad (27)$$

The matrices hopefully have obvious dimensions, and we can initialize the first hidden state vector h_0 however we like, such as a vector of all zeros. This is simply one choice, though. We can imagine many types of choices, such as a vanilla RNN unit or an LSTM unit. The key point is that the hidden state vectors

$$H = \{ h_1, h_2, \dots, h_{T_x} \} \quad (28) \quad H = \{h_1, h_2, \dots, h_{T_x}\} \quad (28)$$

carry forward information from previous words in the sequence via these recurrent connections, much like a [hidden Markov model \(Baum & Petrie, 1966\)](#). A powerful consequence of this model is that RNNs do not limit the size of the input context window. Different input sequences XX can be different sizes, unlike in the N -gram model or in Bengio's model (Equation 1414). See Andrej Karpathy's excellent blog post, [The unreasonable effectiveness of recurrent neural networks](#), for a more detailed presentation of RNNs.

Finally, we define the *context vector* cc as some function of the hidden states:

$$c = q(H). \quad (29) \quad c = q(H). \quad (29)$$

Notice that cc does not have a time index, because it compresses all the temporal information in the input sequence XX into a fixed-width vector. The easiest definition of cc is simply as the last hidden state vector or $c = h_{T_x}$ $c = h_{T_x}$.

This context vector becomes an input to the decoder, another RNN with recurrence relation

$$s_t = f_{\text{dec}}(s_{t-1}, y_{t-1}, c), \quad (30) \quad s_t = f_{\text{dec}}(s_{t-1}, y_{t-1}, c), \quad (30)$$

and hidden states

$$S = \{ s_1, s_2, \dots, s_{T_y} \}. \quad (31) \quad S = \{s_1, s_2, \dots, s_{T_y}\}. \quad (31)$$

The decoder then outputs the sequence YY , one word at a time. The typical objective of a sequence-to-sequence model is again the autoregressive objective of next-word prediction: maximize a log likelihood, in which each conditional probability is modeled via the decoder RNN:

$$\log p(Y) = \sum_{t=1}^{T_y} \log p(y_t | y_{1:t-1}) = \sum_{t=1}^{T_y} \log f_{\text{dec}}(s_{t-1}, y_{t-1}, c). \quad (32)$$

$$\log p(Y) = \sum_{t=1}^{T_y} \log p(y_t | y_{1:t-1}) = \sum_{t=1}^{T_y} \log f_{\text{dec}}(s_{t-1}, y_{t-1}, c). \quad (32)$$

Again, this might be a bit abstract. So for example, one possible instantiation of gg is as a linear transformation of the input variables:

$$f_{\text{dec}}(s_{t-1}, y_{t-1}, c) = W_{zs} s_t + W_{zy} y_{t-1} + W_{zc} c. \quad (33) \quad f_{\text{dec}}(st-1, yt-1, c) = W_z s_t + W_y y_{t-1} + W_z c. \quad (33)$$

Of course, this is just one choice.

Then all the model weights are learned end-to-end by optimizing this log likelihood (Equation 3232). In this way, we can convert a variable-length input XX into a variable-length output YY .

This RNN encoder–decoder framework is powerful, since many problems in NLP can be framed in this way. For example, text summarization, machine translation, and agentic conversation can all be framed as a sequence-to-sequence modeling challenge. To be clear, other researchers around this time had attempted other approaches to handling variable-length sequences, such as the recursive neural tensor network in *Recursive deep models for semantic compositionality over a sentiment treebank* ([Socher et al., 2013](#)). But the RNN encoder–decoder would become the de facto framework of choice for a large range of NLP tasks.

As an aside, sometimes these models are called *sequence transduction models* or *transduction models* or even just *transducers*. My understanding is that “transduction” here just means converting one sequence into another by learning a conditional distribution $p_{\theta}(y_{1:T} | x_{1:S})$. In this context, “transduction” does not have the sense that Vladimir Vapnik gave it. In Vapnik’s definition, transduction loosely means classification of a specific example rather than a general rule for classifying future examples ([Gamerman et al., 2013](#)). But this is not the sense which people mean when they refer to models like the transformer as a “transducer”.

Adaptive context

In my mind, Kalchbrenner, Cho, and Sutskever’s three papers ([Kalchbrenner & Blunsom, 2013](#); [Cho et al., 2014](#); [Sutskever et al., 2014](#)) were the foundations of sequence-to-sequence modeling, and many other papers have built around and off this core idea. But the key point for us here is that these three papers make the same logic choice: they lift the idea of a fixed-length embedding for words or phrases into the context vector cc of a sequential model, such that the models can now support variable-length inputs and outputs and long-range dependencies in each.

However, a problem with this approach was that long-range dependencies got “lost” in this context vector. For example, imagine we had a very long English language text that we wanted to translate into Chinese. Even if our encoder LSTM was good at capturing long-range dependencies in the English sentence, it would be forced to compress that information into a much shorter, fixed-width vector with no temporal structure that would then be fed into the decoder.

This effect was observed by Cho et al in *On the properties of neural machine translation: encoder–decoder approaches* ([Cho et al., 2014](#)). In this paper, the authors write:

Our analysis shows that the performance of the neural machine translation model degrades quickly as the length of a source sentence increases.

And later:

The most obvious explanatory hypothesis is that the fixed-length vector representation does not have enough capacity to encode a long sentence with complicated structure and meaning.

The authors test this hypothesis through a variety of experiments. For example, in one experiment, they report the [BLEU score](#) for an RNN encoder–decoder as a function of sequence length, and they show that the model’s performance degrades as the sentences become longer. So the RNN decoder–encoder was promising, but the fixed-width context vector was a bottleneck on modeling long-range dependencies.

Then in 2014, a seminal paper was published that addressed this problem, *Neural machine translation by jointly learning to align and translate* ([Bahdanau et al., 2014](#)). The main invention of this paper was to use the well-known *attention mechanism* to attend to this context vector. However, the authors barely use the word “attention” in the paper. Instead, they seem to conceptualize it more as a search problem:

In this paper, we conjecture that the use of a fixed-length vector is a bottleneck in improving the performance of this basic encoder–decoder architecture, and propose to extend this by allowing a model to automatically (soft-)search for parts of a source sentence that are relevant to predicting a target word, without having to form these parts as a hard segment explicitly.

I say this paper is “seminal” because, at least to my knowledge, it was really the first paper to use a differentiable attention layer in the rapidly-growing field of NMT. To be clear, the attention mechanism was already known and used outside of NLP. For example, see *Learning to combine foveal glimpses with a third-order Boltzmann machine* ([Larochelle & Hinton, 2010](#)), *Learning where to attend with deep architectures for image tracking* ([Denil et al., 2012](#)), or *Recurrent models of visual attention* ([Mnih et al., 2014](#)). These were all papers that were published between 2010 and 2014 and that applied an attention mechanism to a neural network computer vision system. However, to my knowledge, Bahdanau was the first paper to successfully use attention in NLP. To quote *Effective approaches to attention-based neural machine translation* ([Luong et al., 2015](#)):

In the context of NMT, Bahdanau et al... has successfully applied such attentional mechanism to jointly translate and align words. To the best of our knowledge, there has not been any other work exploring the use of attention-based architectures for NMT.

All that said, “jointly align and translate” is pretty vague, so let’s get technical. Bahdanau’s solution to this bottleneck was to allow each hidden state vector in the decoder to pay attention to possibly all the hidden state vectors in the encoder. What do I mean by “pay

attention to”? Here, each decoder hidden state variable s_i depends not only on the previous hidden state and previous word but also on its own context vector, which is a weighted combination of the encoder’s hidden states!

$$s_i = f_{\text{dec}}(s_{i-1}, y_{i-1}, c_i),$$

$$c_i = \sum_{j=1}^{T_x} \alpha_{ij} h_j. \quad (34) \quad s_i = f_{\text{dec}}(s_{i-1}, y_{i-1}, c_i), \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{j=1}^{T_x} \exp(e_{ij})}$$

This is the main idea of the paper. Each decoder hidden state s_i has access to *all* the hidden states in the encoder via this context vector c_i (Figure 7).

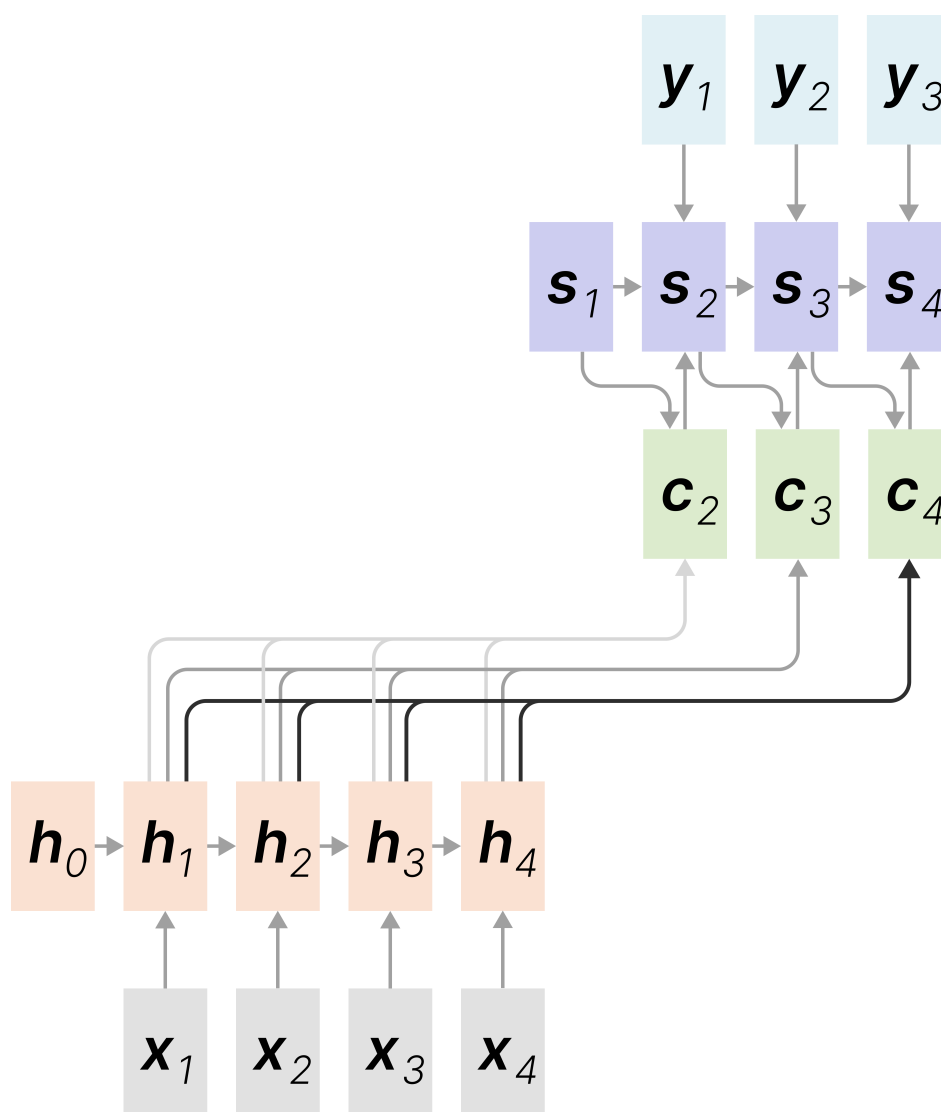


Figure 7. Diagram of the relationship between the encoder hidden states h , the decoder hidden states s , and context vectors in the RNN encoder–decoder with attention proposed by (Bahdanau et al., 2014). Each context vector c_i can attend to any hidden state vector in the encoder to align it with the previous hidden state in the decoder, s_{i-1} . This context vector c_i is then fed into the recurrence relation for s_i .

We can finally define the attention mechanism! Here, it is the weighted sum of hidden state vectors, as this allows each s_i to attend to different parts of the input sequence through its hidden state. Each weight α_{ij} is a linear function of the previous decoder hidden state s_{i-1} and the current decoder hidden state h_j :

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^T \exp(e_{ik})} \quad (35) \quad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^T \exp(e_{ik})}, \quad e_{ij} = v_a^\top z_{ij}, \quad z_{ij} = \tanh(W_a s_{i-1} + U_a h_j).$$

$z_{ij} = \tanh(W_a s_{i-1} + U_a h_j).$ (35)

Let's call α_i the an *alignment vector*, which we infer one per step at a time during the decoding process.

So z_{ij} can be viewed as a shared hidden state, capturing nonlinear information about both the input and output sequence. Importantly, there is one such vector for each input-output pair. And for a given decoder hidden state, the model can up or downweight the relationship to h_j via the parameters v_a . The neural network learns all these model parameters end-to-end via back-propagation, maximizing the log likelihood in Equation 32.

So that's it. As I understand it, ([Bahdanau et al., 2014](#)) was really the first paper to use attention in neural machine translation and probably the most successful use of attention in NLP at the time. The method worked surprisingly well. To quote the paper's conclusion:

Perhaps more importantly, the proposed approach achieved a translation performance comparable to the existing phrase-based statistical machine translation. It is a striking result, considering that the proposed architecture, or the whole family of neural machine translation, has only been proposed as recently as this year.

As an aside, they actually use a bidirectional RNN for the encoder and then concatenated the forward and backward hidden states. But I don't think that adds much to our story or to intuition, and it would muddy Figure 77. The key point is that it was the attention mechanism that allowed for the long-range dependencies encoded by the RNN to be captured through an adaptive context vector. Hopefully, we can now see why the paper uses the words "align and translate". Here, *alignment* really means allowing the model to uncover which parts of the input sequence matter to each part of the output sequence—and it does this via the attention mechanism.

Finally, while writing this blog post, I came across this incredible comment by [Edward Grefenstette](#), published on 3 May 2014:

By and large, the case for deep learning in language hasn't been fully made. It works well for vision and speech, but that doesn't entail that it would carry to semantics. Some excellent shallow models without non-linearities, like the Mnih and Hinton log-bilinear models, are excellent and can be trained very quickly. It's a problem with much "deep learning" work in NLP these days that shallow baselines are never considered or compared to. Deep learning is fascinating and will certainly have an impact in NLP, but don't rush to believe that it's the best solution for your NLP problems.

I love this comment because it is a time-capsule, perfectly capturing how experts in the field felt about neural networks at the time. (Note that Grefenstette has published papers with other researchers in this story, such as Kalchbrenner and Graves.) So even around the time that Bahdanau et al were publishing groundbreaking work on RNN encoder–decoders with attention, deep learning had still not fully proven itself to the community.

Types of attention

The attentive reader might be wondering: wasn't the argument around log-linear models that they were simple and therefore scalable? But Bahdanau's RNN encoder–decoder with attention seems anything but simple. So on some level, yes, Bahdanau's model was a step backwards in terms of complexity. But on another level, it was a proof-of-concept that the attention mechanism worked. (Also, Moore's law.) So researchers quickly built on Bahdanau by studying simpler models and simpler types of attention.

Perhaps the most important paper to directly build on Bahdanau's model was [\(Luong et al., 2015\)](#). In this paper, the authors simplified the model used by Bahdanau, proposed several alternative forms of attention, and showed that an ensemble of attention-based methods produced state-of-the-art results on neural machine translation problems. To be clear, Bahdanau had shown that attention worked and that it seemed to address problems in translating longer sentences, but it did not demonstrably beat the state-of-the-art. Luong's results more directly suggested that attention might be the way forward. So before we get to the transformer, let's understand the attention mechanism better through the lens of this paper.

The first dimension along which we can define attention is local versus global attention. For example, in the attention mechanism in an RNN encoder–decoder, the conceptual lynchpin is that at each time step $i \in \{1, \dots, T_y\}$ in the decoding phase, we construct a context vector c_i which summarizes information from the source sentence via the encoder's hidden states:

$$c_i = \sum_{j=a}^b \alpha_{ij} h_j. \quad (36) \quad c_i = \sum_{j=a}^b \alpha_{ij} h_j. \quad (36)$$

But now I don't precisely define the limits of the sum, a and b . If $a = 1$ and $b = T_x$, then the context vector is constructed by considering all the hidden states of the source sentence. This is what Luong calls *global attention* (Figure 88, left), since each word in the target sentence has access to information about all the words in the source sentence. But we could also define a and b such that they form a window around the decoder's hidden state or model the left-to-right structure of many natural languages. This is what Luong calls *local attention* (Figure 88, middle). So these are two ways in which we can construct the context vector c_i .

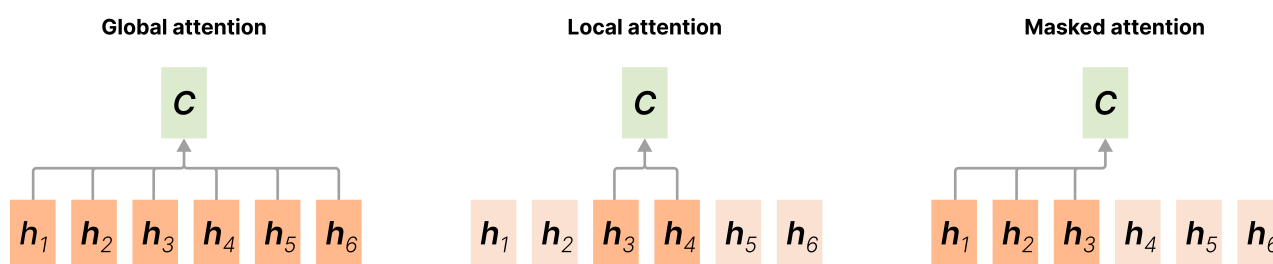


Figure 8. (Left) Global attention is when the attention mechanism considers all possible variables of interest. (Middle) Local attention is when the attention mechanism only considers a subset of possible variables. (Right) Masked attention is a kind of local attention and typically refers to not attending to forward-looking information such as hidden states later in the sequence.

The second dimension along which we can define attention is how we define the alignment weights α_i . For example, the simplest choice is simply that α_i is a one-hot vector, such that c_i selects a single encoder hidden state vector h_k to use in the i -th decoding step. This would be hard- rather than soft-search. But more generally, we can write these alignment weights as the unnormalized output of a *score function*. Using the notation from Equation 3535 above, we can write this as:

$$e_{ij} = \text{score}(h_j, s_{i-1}). \quad (37)$$

And in Luong, the authors explore three main scoring functions. These are *dot-product attention*, *general attention*, and *additive attention*, defined as:

$$e_{ij} = \text{score}(h_j, s_{i-1}) = \begin{cases} h_j^\top s_{i-1} & \text{dot,} \\ h_j^\top W_a s_{i-1} & \text{general,} \\ v_a^\top \tanh(W_a h_j + U_a s_{i-1}) & \text{additive (Bahdanau).} \end{cases} \quad (38)$$

$$e_{ij} = \text{score}(h_j, s_{i-1}) = \begin{cases} h_j^\top s_{i-1} & \text{dot,} \\ h_j^\top W_a s_{i-1} & \text{general,} \\ v_a^\top \tanh(W_a h_j + U_a s_{i-1}) & \text{additive (Bahdanau).} \end{cases} \quad (38)$$

Of course, you can imagine many other score functions. My own view is that it's too difficult here to reason about which form of attention is better in some theoretical sense. Which form works best is an empirical result. In [\(Luong et al., 2015\)](#), the empirical results were mixed in the sense that all three score functions worked well. In fact, the results weren't even strong enough for the authors to claim that attention-based methods were demonstrably better. This was their conclusion:

Our analysis shows that attention-based NMT models are superior to non-attentional ones in many cases, for example in translating names and handling long sentences.

So by late 2015, just two years before the transformer, attention was just becoming popular in NMT but was not yet the de facto modeling choice.

That said, obviously this will change, and when it does, there will be a clear winner amongst the choices above, and that winner is dot-product attention. Dot-product attention is the variant used by the transformer, and thankfully, in my mind it is the most intuitive since the dot product is a standard way to measure the [similarity between two vectors](#). So we can interpret the dot-product score function as measuring the similarity between the encoder and decoder hidden states.

The third and final dimension along which we can define attention is through the variables of interest. In order to understand what I mean, we can no longer refer to attention in terms of hidden states of RNNs. We need more general terminology. In the literature, attention is often viewed through the lens of information retrieval. In this literature, a *query* is what you are asking for; a *key* is what you can search through; and a *value* is what you can return.

Queries	Keys	Values
"indian food near me"	cuisine: Tibetan	Zaab Zaab
"isan thai"	region: Puebla	Lhasa Fast Food
"best momos jackson heights"	neighborhood: Woodside	Angel
	price: \$\$	El Toum
	rating: 4.5 stars	

Figure 9. An illustration of queries, keys, and values using information retrieval from a data source about restaurants. The queries are user search terms; the keys are metadata against which we weight the results; and the values are the returned results.

Let me give an example (Figure 99). Imagine I type some text into a search bar: "indian food near me". This text is the query. Now imagine the search engine runs that query against a bunch of metadata associated with different restaurants. For example, restaurant descriptions, keywords, reviews, ratings, and distances from my location.

These metadata are the *keys*. So the query is “run against” the keys. Finally, the thing returned are candidate restaurants. These are *values*. In the language of information retrieval, we can describe the attention mechanism as a kind of soft-search, since it can return a linear combination of the values. As you may recall, this is precisely how Bahdanau described their model in the quote above.

So in Bahdanau’s RNN encoder–decoder, the decoder’s hidden states s_i are the queries, since for each hidden state s_i we want to search through the source sentence. The encoder’s hidden states h_j are the keys, since these are the metadata associated with the source sentence that we can search through. Finally, the encoder’s hidden states are also the values, since the context vector c_i is a weighted combination of these encoder hidden states. This language is useful because it disambiguates the attention mechanism from a specific choice of model and even from which variables in that model are being used for what.

Now that we understand this terminology, we can express ourselves more cleanly and abstractly. And with this terminology, it becomes clear that the keys, queries, and values need not be different objects in our model at all! In fact, queries, keys, and values can all be taken from the same set. For example, imagine we have a model with a hidden state h . This is not necessarily the hidden state of an RNN or even a sequential model. We could define a kind of attention such that the queries (q), keys (k), and values (v) are all functions of this hidden state:

$$\begin{aligned} q_i &:= f_q(h_i), \\ k_j &:= f_k(h_j), \\ v_j &:= f_v(h_j), \\ \alpha_{ij} &= \text{softmax}(\text{score}(q_i, k_j)), \sum_j \alpha_{ij} = 1, \\ c_i &= \sum_j \alpha_{ij} v_j. \end{aligned} \quad (39) \quad \begin{aligned} q_i &:= f_q(h_i), \\ k_j &:= f_k(h_j), \\ v_j &:= f_v(h_j) \end{aligned}$$

$$), = \text{softmax}(\text{score}(q_i, k_j)), \sum_j \alpha_{ij} = 1, = \sum_j \alpha_{ij} v_j. (39)$$

This is obviously different from the attention mechanism in Bahdanau. In Bahdanau, the authors use *cross-attention*, which is attention where the queries come from one set and the keys and values come from a different set. As you can imagine, typically the keys and values come from the same set, although they might have their own maps or projections such that they are correlated but not identical. For example, we might run a query against restaurants (keys) and also return restaurants (values). However, *self-attention* is when the queries, keys, and values all come from the same set of variables! To continue abusing our running example, we essentially compute the similarity between restaurants of interest and restaurants we have data about, and then use those weights to return a weighted combination of restaurants!

To my knowledge, the first paper to use self-attention in NLP was *Long short-term memory-networks for machine reading* ([Cheng et al., 2016](#)). This model is a bit complicated, and I don't think it's that important to understand here. The key point is only to grok that attention does not have to be cross-attention as in Bahdanau. Instead, we can have a sequence attend to itself to decide what parts of the sequence matter—or self-attention! This is how this idea was described in the paper:

A remaining practical bottleneck for RNNs is memory compression ([Bahdanau et al., 2014](#)): since the inputs are recursively combined into a single memory representation which is typically too small in terms of parameters, it becomes difficult to accurately memorize sequences ([Zaremba & Sutskever, 2014](#)). In the encoder-decoder architecture, this problem can be sidestepped with an attention mechanism which learns soft alignments between the decoding states and the encoded memories ([Bahdanau et al., 2014](#)). In our model, memory and attention are added within a sequence encoder allowing the network to uncover lexical relations between tokens.

The important phrase here is “within a sequence encoder”. Here, the attention is not applied across the encoder and decoder but rather is applied as intra- or self-attention within the encoder.

So circa 2017, attention was being studied in its many forms: local versus global, additive versus multiplicative, and cross versus self. And it was being more widely used in NLP, with papers like *A structured self-attentive sentence embedding* ([Lin et al., 2017](#)) and *Bidirectional attention flow for machine comprehension* ([Seo et al., 2016](#)). That said, I do not think any specific form was clearly the dominant one. Rather, each showed promise in its own way. For example, in March 2017, Google Brain published *Massive exploration of neural machine translation architectures* ([Britz et al., 2017](#)). This was published just months before the transformer would be published, and even here, attention is only a minor player. In that paper's conclusions, the authors list six main results, and the only one about attention is a single sentence:

Parameterized additive attention yielded the overall best results.

Notice that additive attention is not even the form of attention used by the transformer!

So at least as best as I understand it, attention was well-understood and widely-studied in 2017, but it was by no means considered the main ingredient or the next logical step. Many researchers were still pushing the limits of training RNNs at scale, rather than trying other approaches. See *Exploring the limits of language modeling* ([Jozefowicz et al., 2016](#)) for example. However, in June 2017, all that was about to change. The transformer's time had come.

The transformer

In 2017, researchers at Google Brain published *Attention is all you need* ([Vaswani et al., 2017](#)), which is the original paper introducing the transformer architecture. This was their proposal, which I hope now makes sense given the context so far:

The dominant sequence transduction models are based on complex recurrent or convolutional neural networks that include an encoder and a decoder. The best performing models also connect the encoder and decoder through an attention mechanism. We propose a new simple network architecture, the Transformer, based solely on attention mechanisms, dispensing with recurrence and convolutions entirely.

The authors acknowledge that the sequence-to-sequence framework with neural networks was state-of-the-art, and they specifically call out the RNN encoder–decoder architecture with attention from Bahdanau, Luong, and others. Their proposal is simple: keep the encoder–decoder framework but replace everything else with attention.

How might someone have come to this idea at the time? Why would it be a good idea to try? Their observation is that the sequential nature of RNNs inhibits training these models at scale:

Recurrent models typically factor computation along the symbol positions of the input and output sequences. Aligning the positions to steps in computation time, they generate a sequence of hidden states h_t , as a function of the previous hidden state h_{t-1} and the input for position t . This inherently sequential nature precludes parallelization within training examples, which becomes critical at longer sequence lengths, as memory constraints limit batching across examples.

Their proposal is to use attention rather than RNNs to uncover dependencies within the input and output sequences. This is a good idea to try not because attention is obviously better than recurrence per se. It's that attention is parallelizable! They write:

The Transformer allows for significantly more parallelization and can reach a new state of the art in translation quality after being trained for as little as twelve hours on eight P100 GPUs.

We have seen this before. Recall how the unlock for word embeddings in ([Mikolov et al., 2013](#); [Mikolov et al., 2013](#)) was simplifying the models and focusing on scale. But then the RNN encoder–decoder architecture in ([Bahdanau et al., 2014](#)) with attention took us backwards in terms of model complexity. So the transformer is a similar story: take the best modeling ideas, strip them down, and train the simplified model at scale.

That's it. Properly understood in context, the transformer is a modest conceptual leap from the existing literature. My point is not that the transformer is “obvious” in the sense that it is not an impressive invention. My point is to demystify the research product by

underscoring the process. In context, the transformer should make sense as something someone might have tried in 2017.

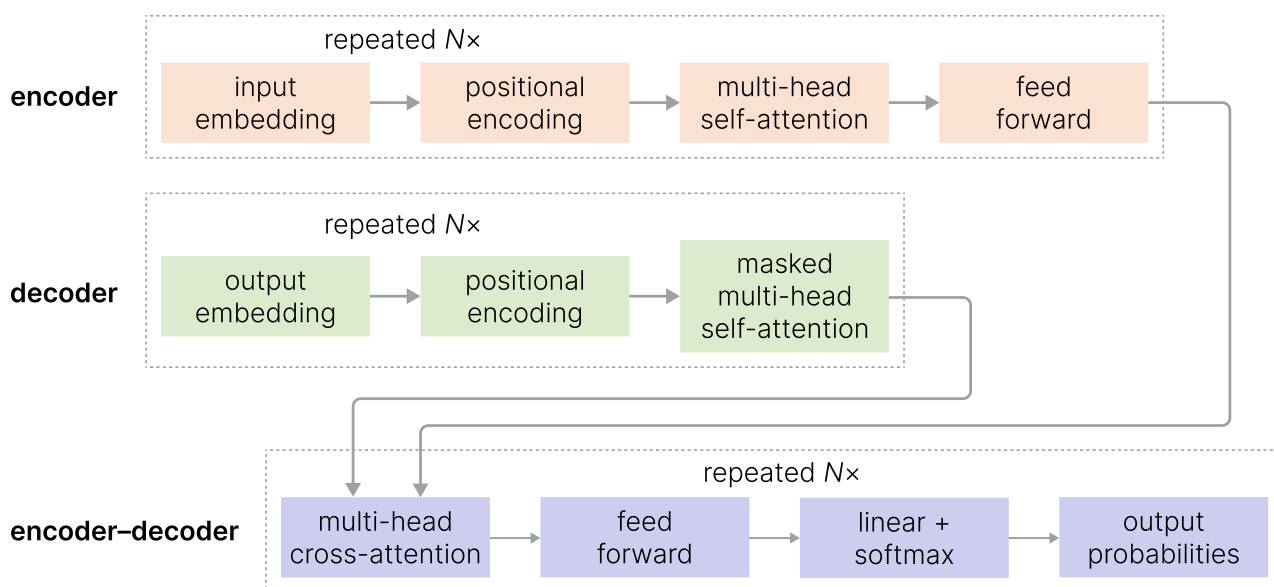


Figure 10. The transformer model architecture. Both the encoder and decoder use positional encoding and multi-head self-attention. The decoder uses masked attention to maintain the left-to-right structure of most natural languages. The encoder-decoder uses cross-attention a la [\(Bahdanau et al., 2014\)](#) to align the hidden states before outputting probabilities.

The model architecture might look intimidating, but it is pretty straightforward when viewed in the right context (Figure 10). At a high level, the transformer is an encoder-decoder, with two big kinds of attention. First, we have cross-attention between the outputs of the encoder and the inputs to the decoder. This is completely analogous to the cross-attention in Bahdanau and others. But then we also have self-attention within the decoder and encoder. This completely replaces the recurrence relations of RNNs. Finally, the model uses something called *positional encoding*, which I'll define shortly, to handle the fact that attention is not naturally sequential a la an RNN. Everything else is details. For example, the transformer also uses layer normalization [\(Ba et al., 2016\)](#) and residual connections [\(He et al., 2016\)](#), but these are not unique or novel contributions. Even multi-head attention is not conceptually hard. So understood in context, the transformer is pretty straightforward. Let's go through the main bits in detail.

First, positional encoding. A key challenge for the attention mechanism is that it does not inherently capture sequential structure. Thus, the relative positions of words in a sequence can be easily lost. In Vaswani, the authors propose attaching vectors of numbers to the inputs to capture this position-dependent information. The precise functional form of these numbers doesn't really matter to us. The point is that we're encoding the position of each word so that we can still model the sequential structure of natural language.

After adding position-dependent information, the transformer encodes the input sequence. But rather than passing the data through an RNN, it passes the data through multi-head attention layers. We'll discuss "multi-head" in a moment, but the basic attention mechanism is what the authors call *scaled-dot product attention*. Let's define it. Let $Q \in \mathbb{R}^{M \times D_k}$ be a matrix of queries, let $K \in \mathbb{R}^{N \times D_k}$ be a matrix of keys, and let $V \in \mathbb{R}^{N \times D_v}$ be a matrix of values. Then scaled dot-product attention is:

$$\text{attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{D_k}}\right)V. \quad (40)$$

(40)

When I first read Vaswani, I had not yet read Bahdanau or Luong, and thus I was completely confused by Equation 40. It was not at all obvious what any of these values represented or why any of this machinery worked. And the paper itself gave a pretty opaque explanation:

An attention function can be described as mapping a query and a set of key-value pairs to an output, where the query, keys, values, and output are all vectors. The output is computed as a weighted sum of the values, where the weight assigned to each value is computed by a compatibility function of the query with the corresponding key.

Without context, this explanation is not very helpful. However, armed with a better understanding of attention, we can make sense of this. In the cross-attention between the encoder and decoder, the queries are analogous to the hidden states of the RNN decoder, while the keys and values are analogous to the hidden states of the RNN encoder. And if we remove the sample dimension (so let $N = 1$), we can rewrite Equation 40 in a way that looks like the types of attention in Equation 38:

$$\begin{aligned} \text{score}(q_i, k_j) &= e_{ij} = \frac{q_i^T k_j}{\sqrt{D_k}}, \\ \alpha_{ij} &= \frac{\exp(e_{ij})}{\sum_{k=1}^{D_v} \exp(e_{ik})}, \\ \text{attention}(\alpha_i, v_i) &= \sum_{k=1}^{D_v} \alpha_{ik} v_i. \end{aligned} \quad (41)$$

$$q_i^T k_j = \sum_{k=1}^{D_v} \exp(e_{ik}) \exp(e_{ij}), \quad \alpha_{ij} = \frac{\exp(e_{ij})}{\sum_{k=1}^{D_v} \exp(e_{ik})}, \quad \text{attention}(\alpha_i, v_i) = \sum_{k=1}^{D_v} \alpha_{ik} v_i. \quad (41)$$

So this is identical to the multiplicative or dot-product attention proposed in Luong (Equation 3838), modulo a scaling factor $\sqrt{D_k}$

. In Equation 4040, we are just packaging it into a matrix form so that we can compute this attention over many samples at once. In other words, this is a highly parallelizable version of the dot-product attention.

I think one of the reasons the transformer can be confusing is the use of two types of attention and the generic language of queries, keys, and values, whose definitions change depending on the type of attention. In the encoder, the transformer uses self-attention. So the query represents the current vector in the input sequence, while the keys and values are all the other vectors in the input sequence. And in the decoder, the query represents the current vector in the output sequence, while the keys and values are all the other vectors in the output sequence—modulo masking, which I'll mention in a moment. Finally, the attention between the encoder and decoder (in the paper, Vaswani calls this “encoder–decoder attention”), the query is the current vector in the decoder output (analogous to s_i in the RNN encoder–decoder), while the keys and values are the encoder's hidden outputs (analogous to h in the RNN encoder–decoder).

Note that “masked” in “masked multi-head self-attention” just refers to a masking out of words in the decoder's self-attention mechanism. This is because attention has no inherent sequential structure like RNNs. So we have to enforce this by masking regions of the output sequence (Figure 88, right). This allows the transformer to be trained in the standard autoregressive framework we have discussed since (Bengio et al., 2003).

Finally, the transformer learns multiple sets of parameters associated with the attention mechanism at once. This is what the paper calls *multi-head attention*. Instead of having a single attention function, we can run multiple attention functions in parallel, say A times. By way of analogy, recall that in the RNN encoder–decoder, we had the following attention parameters (Equation 3535):

$$\{W_a, U_a, v_a\}. \quad (42) \quad \{W_a, U_a, v_a\}. \quad (42)$$

In Bahdanau (Equation 3535) the subscript aa just denotes that these are attention-related weights. It is not actually indexing into multiple such weights (that is, $A = 1$). But we *could* do that. We could say that aa is indexing into different parameters, $a \in \{1, 2, \dots, A\}$. This would have made Bahdanau's model slower to train, but it would have allowed for multiple cross-attention mechanisms to be learned at once. In Bahdanau, they don't actually do this, likely because it's too expensive! The precise details are different in Vaswani, but this is all multi-head attention is in theory. It is multiple parallel attention mechanisms.

So that's it. That's the transformer. The results were impressive. To be clear, it was not an AlexNet moment, but the results were clearly better than benchmarks and more importantly, the model was way more efficient. For example, one of the benchmarks in

Vaswani is the ConvS2S Ensemble from *Convolutional sequence to sequence learning* ([Gehring et al., 2017](#)). The idea of this paper is similar to the transformer: train a bigger sequence-to-sequence model by eschewing recurrent connections in favor of parallelizable convolutional layers. In both English-to-German and English-to-French translation, the transformer beats this model in BLEU score. But more importantly, it is more efficient. For example, according to Vaswani, the ConvS2S Ensemble required 1.2×10^{21} 1.2×10^{21} flops to train their English-to-French model, whereas the transformer required 3.3×10^{18} 3.3×10^{18} flops. So the transformer had comparable results with a 360x reduction in flops! In my mind, this the real insight. It is not that attention is absolutely the best way to model the problem. Rather, the transformer is on the Pareto frontier between modeling the problem well enough and being scalable enough.

To see the transformer in code, see Sasha Rush's excellent [The annotated transformer](#).

Generative pre-training

The transformer was a revolutionary architecture, and explicitly designed to scale. However, in reality the original model was tiny by today's standards. The biggest variant only had 2.13 million parameters, and the largest dataset it was trained on, the WMT 2014 English–French datasets, only had 36 million sentences. But the paper proved that the transformer worked well as a generic transduction model.

However, despite the paper's name, the transformer architecture was not enough. Researchers also needed advancements in how these models were trained in order to make the commodity LLMs most people interact with today. To simplify the discussion, I'll focus on training for OpenAI's GPT series. My understanding is that OpenAI made a lot of the big contributions here, and so their papers are good landmarks to follow. Loosely, the three training stages they discuss in their GPT papers are generative pre-training, discriminative fine-tuning, and reinforcement learning with human feedback. Let's work through the first two in detail here and the last one in detail in the next section.

In 2018, roughly a year after the transformer was published, OpenAI published *Improving language understanding by generative pre-training* ([Radford et al., 2018](#)). The main idea of the paper is to pre-train a transformer with as much unlabeled data as possible before fine-tuning it with task-specific supervised training. In the paper, the authors call the first step *generative pre-training* and the second step *discriminative fine-tuning*. (The words “generative” and “discriminative” have a long history in machine learning; see ([Ng & Jordan, 2001](#)) for a discussion.) As the OpenAI paper title suggests, the key focus was on generative pre-training. Supervised learning obviously matters, but the idea was that one could use unsupervised training at scale to build a base model and then use supervised learning to train more task-specific downstream models.

Let's look at generative-pretraining in a bit more detail. Since we do not have labels, we need some way to formalize the problem. In generative pre-training, the objective is *next-word prediction* as in the autoregressive framework. In other words, the objective is maximum likelihood estimation on Equation 11:

$$L_{\text{GPT}}(\theta) = \sum_{t=1}^T \log p_{\theta}(w_t | w_{1:t-1}). \quad (43) \quad \text{LGPT}(\theta) = \sum_{t=1}^T \log p_{\theta}(w_t | w_{1:t-1}).$$

(43)

As we saw around Equation 1212, maximum likelihood estimation here is equivalent to minimizing the cross-entropy loss between our model's prediction of w_t and the ground truth. So this whole process is unsupervised, and we can train our model on lots and lots and lots of data.

It's worth observing that Equation 4343 is only one generative pre-training objective function, and it has limitations. In particular, note that the autoregressive framework means that the model is pre-trained "left to right" and thus limits the set of suitable downstream tasks. To address this limitation, in 2019, Google AI published *BERT: Pre-training of deep bidirectional transformers for language understanding* ([Devlin et al., 2019](#)). Here, the authors propose a pre-training objective that learns bidirectional representations. Rather than pre-training using the autoregressive framework, they pre-train using a "masked language model", which randomly masks some of the tokens to predict, without assuming a left-to-right relationship. Quoting that paper:

Unlike left-to-right language model pre-training, the [masked language model] objective enables the representation to fuse the left and the right context, which allows us pre-train a deep bidirectional Transformer.

More formally, let $M \subseteq \{1, 2, \dots, T\}$ be a mask denoting positions in the input sequence $w_1:T$, and let $\neg M$ denote all indices that are *not* in M . The denoising objective is to maximize

$$L_{\text{MLM}}(\theta) = \sum_{i \in M} \log p_{\theta}(w_i | w_{\neg M}). \quad (44) \quad \text{LMLM}(\theta) = \sum_{i \in M} \log p_{\theta}(w_i | w_{\neg M}). \quad (44)$$

This idea was inspired by the [Cloze test \(Taylor, 1953\)](#), and the idea was that this bidirectional transformer can then be fine-tuned on a much wider range of downstream tasks. That said, my understanding is that generative pre-training is fairly standard. The left-to-right assumption is simple and matches natural language, coding, and so forth. But I am not confident about what is used in absolutely state-of-the-art foundation models right now.

Either way, neither objective function is enough. For example, consider a conversational agent built on top of a large language model. Now imagine the user prompts an LLM with the following question: "I am having trouble getting a date. Any advice?" If the LLM is only trained on next-word prediction, a plausible response might be: "You'll never find true love!" From the perspective of the distribution of English words on the internet, this is not an unreasonable response. But it is not helpful and hopefully not true. In other words, next-word prediction is obviously not enough for most meaningful tasks that leverage LLMs. So the second step in training is *discriminative fine-tuning*.

“Discriminative fine-tuning” is just a fancy way of saying supervised learning on specific tasks:

$$L_{\text{DFT}}(\theta) = \sum_{y, x_{1:T}} \log p_{\theta}(y | x_{1:T}). \quad (45) \quad \text{LDFT}(\theta) = \sum_{y, x_{1:T}} \log p_{\theta}(y | x_{1:T}). \quad (45)$$

Here, I am using standard notation for supervised learning (x, y) (x, y) , rather than the notation in this post. There are some possible subtleties here. For example, in the GPT-1 paper, they optimize a weighted objective function to balance between generative pre-training and discriminative fine-tuning:

$$L_{\text{final}} = L_{\text{DFT}} + \lambda L_{\text{GPT}}. \quad (46) \quad L_{\text{final}} = \text{LDFT} + \lambda \text{LGPT}. \quad (46)$$

This ensures that during fine-tuning, the model does not unlearn parameters that are good for next-word prediction.

In the process of trying to fine-tune LLMs, researchers have built ever more task-specific datasets to tackle problems like question-and-answering ([Reddy et al., 2019](#)), text summarization ([Nallapati et al., 2016](#)), commonsense inference ([Zellers et al., 2019](#)), code generation ([Chen et al., 2021](#)), broader discourse context ([Paperno et al., 2016](#)), and grade school math ([Cobbe et al., 2021](#)). A pre-trained LLM can be fine-tuned in a dizzying number of ways.

I have two caveats to the above presentation. First, I want to emphasize that this two-step training procedure was not a conceptual leap for researchers. At the time, researchers were already training models with pre-trained word embeddings, and even before this, this two-step training procedure was both understood and used in practice. For examples, see ([Collobert & Weston, 2008](#); [Ramachandran et al., 2016](#); [Hinton et al., 2012](#)). Furthermore, researchers knew to use both pre-trained word embeddings and to even have task-specific objectives when training their word embeddings. Remember ELMO? The earliest reference I have found to this idea of pre-training—I am sure there are earlier ones—is from the 2006 paper *Greedy layer-wise training of deep networks* ([Bengio et al., 2006](#)). Here, the authors write:

We hypothesize that three aspects of this strategy are particularly important: first, pre-training one layer at a time in a greedy way; second, using unsupervised learning at each layer in order to preserve information from the input; and finally, fine-tuning the whole network with respect to the ultimate criterion of interest.

In these examples above, it’s clear the authors recognize that one can pre-train a model with unsupervised learning and then fine-tune it with supervised learning. So even in the GPT paper, the novel contribution is not generative pre-training per se, but only applying it to language modeling at an unprecedented scale.

My second caveat is that while discriminative fine-tuning is used in commodity LLMs that many people interact with, the early GPT models were remarkable in part because they did not need fine-tuning! For example, as their titles suggest, the GPT-2 paper *Language*

models are unsupervised multitask learners ([Radford et al., 2019](#)) and the GPT-3 paper *Language models are few-shot learners* ([Brown et al., 2020](#)) both focus on massively pre-trained transformers that excel in the zero-shot ([Palatucci et al., 2009](#)) and few-shot settings, on a variety of tasks like reading comprehension, summarization, and translation. For example, in the GPT-3 paper, the authors are explicit:

For all tasks, GPT-3 is applied without any gradient updates or fine-tuning.

That said, many related research projects did fine-tune these models, and the GPT-4 technical report ([Achiam et al., 2023](#)) does discuss post-training alignment, which we'll discuss next. So while each LLM may be trained in slightly different ways, I am fairly confident most foundation models today are trained with some combination of massive pre-training and then optionally task-specific fine-tuning and alignment. I'm sure the precise details vary depending on the final product. For example, OpenAI's [Codex](#) is a version of GPT-5 but optimized for agentic coding.

Alignment

Making LLMs bigger does not necessarily make them better at following a user's intent or make them more aligned with human values. For example, we might not want conversational agents to lie, to make racist jokes, or to sexually harass the user. But nothing in the autoregressive framework accounts for this. We need to somehow encode these human values into the model. For some of these properties, we might be able to use a form of fine-tuning. There are datasets for this, such as the ETHICS dataset ([Hendrycks et al., 2020](#)) or the RealToxicityPrompts dataset ([Gehman et al., 2020](#)). But the limitations here are fairly obvious. And for many human values, it would be difficult to encode because the property itself is hard to define.

To encode these properties, state-of-the-art LLMs are often trained using something called *reinforcement learning with human feedback* (RLHF). RLHF was developed around the same time as the transformer, in *Deep reinforcement learning from human preferences* ([Christiano et al., 2017](#)). The original motivation was how to expand the reinforcement learning (RL) framework beyond problems with well-specified reward functions. For example, RL has been used to great effect to play Go ([Silver et al., 2016](#)), Atari ([Mnih et al., 2013](#)), and Dota 2 ([Berner et al., 2019](#)), but what these tasks have in common is that their reward functions are relatively simple and their environments are relatively easy to simulate. But to borrow two examples from Christiano et al, how would you teach a machine-learning model to clean a table or to scramble an egg? It's hard to come up with an objective function or simulation environment for these kinds of tasks.

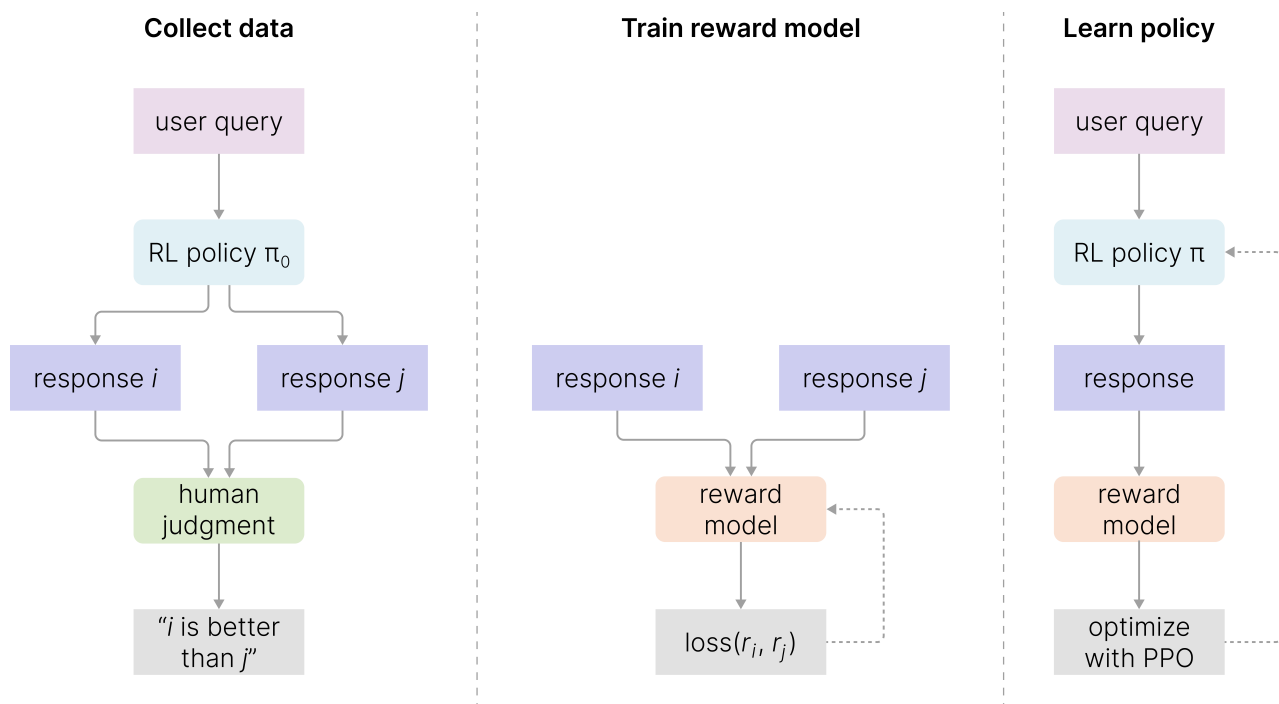


Figure 11. Illustration of the three steps to reinforcement learning with human feedback, inspired by Figure 22 in (Stiennon et al., 2020). (Left) Collect data by generating multiple candidate responses from one or multiple policies; have human labelers rank these responses. (Middle) Train a reward model to prefer human responses. (Right) Train a reinforcement learning policy, using the rewards from the reward model.

What we need, then, is a reward function that can be defined by human feedback and thus by human preferences. Broadly, RLHF is a three-step training procedure (Figure 11). First, humans are used to label a dataset which captures human preferences. For example, if the task is text summarization, the dataset might be different candidate summarizations, with the best summarization being defined by human scorers. Second, researchers train a reward function on these data, which predicts which output the humans would prefer. Finally, given this reward function, researchers can apply standard RL algorithms such as proximal policy optimization or PPO (Schulman et al., 2017) to fine-tune the model.

Fine-tuning LLMs with RLHF is now fairly standard practice. For example, GPT-2 was fine-tuned this way in *Fine-tuning language models from human preferences* (Ziegler et al., 2019), while GPT-3 was fine-tuned this way in *Training language models to follow instructions with human feedback* (Ouyang et al., 2022) and in *Learning to summarize with human feedback* (Stiennon et al., 2020). And the GPT-4 whitepaper (Achiam et al., 2023) states that the model was trained with RLHF.

That said, as the content of this post approaches present day, it is increasingly likely I am writing things that lack nuance. For example, in the GPT-4 whitepaper, the authors write:

The model’s capabilities on exams appear to stem primarily from the pre-training process and are not significantly affected by RLHF.

So while I am confident that generative pre-training is not enough and that certainly large foundation models trained today do more than just pre-training, the precise details of what else goes into which models are both opaque and rapidly changing.

Finally, it's worth mentioning other work on LLM alignment beyond RLHF. In particular, Anthropic has a number of papers on model alignment. For example, the paper *A general language assistant as a laboratory for alignment* ([Askell et al., 2021](#)) focuses on encoding alignment into LLMs, where they define an aligned model as a model that is “helpful, honest, and harmless”. They explore a variety of techniques, such as imitation learning, binary discrimination, and ranked preference modeling. However, the best way to tackle alignment is still an open-ended problem.

Zero to one

Large language models are the result of at least forty years of research, dating back to work by Hinton, Rumelhart, and others on distributed representations in the 1980s. In the early 2000s, Bengio et al introduced the first probabilistic language model using neural networks. However, it wasn't until after AlexNet, nearly a decade later, that researchers were finally able to train neural network language models at scale. They quickly discovered that these distributed representations captured semantic and syntactic structure, even when using simple log-linear models. This idea of word and phrase-level embeddings was then extended to variable-length sequences with long-range dependencies via transduction models, particularly models with an attention mechanism on the hidden states. Finally in 2017, Vaswani et al introduced the transformer, which simplified transduction models by using all attention. In the eight years since, the main advancements have been training these models on more and more data, using techniques such as generative pre-training and reinforcement learning with human feedback.

After learning about how LLMs work, I am reminded of one of my favorite Richard Feynman quotes: [“It is not complicated. It's just a lot of it.”](#) Of course, this is dramatic, but I do think it emphasizes an important point: none of the ideas in this post are terribly complicated. No single idea is beyond the abilities of a smart teenager to understand. But what is beautiful and surprising and remarkable is that the phenomena we observe in LLMs is not magic but simply the emergence of a complex system from simple rules.

Today, LLMs are everywhere, and it's easy to get lost in the models and benchmarks. OpenAI has the GPT series ([Radford et al., 2018](#); [Radford et al., 2019](#); [Brown et al., 2020](#); [Achiam et al., 2023](#)). Google has the Gemini family of models ([Team et al., 2023](#)) as well as PaLM ([Chowdhery et al., 2023](#)), LaMDA ([Thoppilan et al., 2022](#)), Gopher ([Rae et al., 2021](#)), and BERT ([Devlin et al., 2019](#)). Anthropic has the Claude family of models, named in ascending order of size and power: Haiku, Sonnet, and Opus. Finally, Meta has its LLaMA series ([Touvron et al., 2023](#); [Touvron et al., 2023](#)). And there are many, many more, such as open-weight models like DeepSeek-R1 ([Guo et al., 2025](#)), which made

headlines earlier this year. It would be its own blog post to cover the differences between these. But in essence, every model is the same: a large transformer-style model, pre-trained at massive scale using next-word prediction.

The biggest differences have been the size of the training data and the size of the model. For example, GPT-1 is thought to have 117 million parameters (estimated from “Model specifications” in the original paper), while GPT-2 and GPT-3 had 1.5 billion and 1.75 billion parameters respectively—although in [\(Stiennon et al., 2020\)](#), the authors, OpenAI researchers, mention using “large pretrained GPT-3 models with as many as 6.7 billion parameters”. Regardless, there are roughly three orders of magnitude in the number of parameters in just two generations. OpenAI did not publish the model sizes for GPT-4 and GPT-5, and the latter does not even have a whitepaper but only a [“system card”](#). I have not seen published numbers for Google’s large Gemini models, but the smallest model (the nano) has 1.8-3.25 billion parameters [\(Team et al., 2023\)](#). Google DeepMind’s Gopher had 280 billion parameters in 2021, while PaLM had 540 billion parameters in 2022! So industry secrets aside, it is safe to say that large foundation models today are likely pushing into the trillions of parameters. The era of truly large language models has begun.

In my mind, the main counterintuitive result of LLMs is that training ever larger models using primarily next-word prediction is enough to exhibit human-level performance on such a broad range of tasks. And scale truly does matter here. For example, in the GPT-4 technical report, the authors observe that on a simulated bar exam, GPT-4 scored in the 90th percentile, while GPT-3.5 scored in the 10th. Or consider chain-of-thought reasoning [\(Ling et al., 2017\)](#), which is a new way of prompting LLMs in order to improve their reasoning by forcing them to explain each step. In *Chain-of-thought prompting elicits reasoning in large language models* [\(Wei et al., 2022\)](#), the authors write:

Chain-of-thought prompting does not positively impact performance for small models, and only yields performance gains when used with models of ~100B parameters. We qualitatively found that models of smaller scale produced fluent but illogical chains of thought, leading to lower performance than standard prompting.

So you don’t get anything useful from chain-of-thought reasoning until you have a model that is roughly 50,000 times the size of the original transformer.

Why does scaling work? I don’t think anyone knows. But it is an effect that has been observed repeatedly since AlexNet, in the above works and also in meta-analyses such as *Scaling laws for neural language models* [\(Kaplan et al., 2020\)](#), *Scaling language models: methods analysis, and insights from training Gopher* [\(Rae et al., 2021\)](#), and *Emergent abilities of large language models* [\(Wei et al., 2022\)](#). And this phenomenon was both observed and predicted in the famous blog post [The Bitter Lesson](#). In that post, one of the pioneers of RL, Richard Sutton, argues that the “bitter lesson” from AI history is that general, compute-efficient, scalable methods outperform human knowledge and domain-

specific insights. This lesson is bitter because it means that expert labor, clever domain-specific theories, handcrafted features, elegant mathematics and beautiful algorithms—these all get dwarfed and outpaced by brute-force search and learned representations.

As a stark example of this, consider the observation that early LLMs were bad at mathematics ([Hendrycks et al., 2021](#)). But today, state-of-the-art models are now [winning gold at the International Math Olympiad](#), and Terry Tao has [compared o1 to a mediocre graduate student](#). The rate of change is immense. By “early LLMs”, I am referring to models from five years ago, and the transistor is only a hundred years old. Did you know that a modern graphics card can perform [36 trillion calculations a second](#)? Moore’s law and all that.

If you feel that it’s a bit perverse that next-word prediction is a sufficient objective to solve elite math problems, if this feels like a stochastic parrot outsmarting you, then you might feel some of the discomfort early linguists felt at statistical language modeling. This is the visceral feeling of the bitter lesson. Our specialized knowledge feels expendable and our intuitions about understanding seem irrelevant in the face of raw computation and speed.

But my own view—since you’ve read this far—is that for the time being, machine learning systems are powerful tools that can still be combined with real expertise. Perhaps the best example of this is AlphaFold from Google DeepMind, published in *Highly accurate protein structure prediction with AlphaFold* ([Jumper et al., 2021](#)). This model achieved near-experimental accuracy on the protein prediction problem. On the one hand, it did so with black-box deep learning. On the other hand, the work leaned heavily on prior biological art, for example using sequences from evolutionarily related proteins and 3D coordinates of homologous structures as inputs to the model. It clearly sidestepped Levinthal’s combinatorial search, even if we do not know how.

So what happens next? Even the world’s leading experts can disagree. But in my mind, if anyone deserves the last word here, it is Geoff Hinton, who has been a contrarian believer in neural networks since the 1980s and who, along with Yoshua Bengio and Yann LeCun, won the [Turing Prize in 2018](#) in part for work mentioned in this post. In a 2024 [BBC interview](#), Hinton argued that LLMs do in fact understand natural language and that they are our current best theory of how the brain understands language as well. In his view, it is only a matter of time before LLMs exceed human intelligence. Certainly, by some metrics and along some dimensions, they already have.

Resources

Below are some additional resources, which I found useful or interesting while writing this post:

- 3Blue1Brown: [Inside an LLM](#)
- Stefania Cristina: [The Bahdanau attention mechanism](#)
- Stefania Cristina: [The attention mechanism from scratch](#)

- Dan Jurafsky and James H. Martin: [Speech and language processing](#)
 - Andrej Karpathy: [The unreasonable effectiveness of recurrent neural networks](#)
 - Andrej Karpathy: [Let's build GPT: from scratch, in code, spelled out](#)
 - Chris Olah: [Understanding LSTM networks](#)
 - Dwarkesh Patel: [Interview with Richard Sutton](#)
 - Sasha Rush: [The annotated transformer](#)
 - Ari Seff: [How ChatGPT is trained](#)
 - Ari Seff: [What are transformer neural networks?](#)
 - StackOverflow: [What exactly are keys, queries, and values in attention mechanisms?](#)
 - Mohammed Terry-Jack: [Deep learning: The transformer](#)
1. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). Attention is all you need. *Advances in Neural Information Processing Systems*, 30.
 2. McCulloch, W. S., & Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4), 115–133.
 3. Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning representations by back-propagating errors. *Nature*, 323(6088), 533.
 4. Hinton, G. E. (1986). Learning distributed representations of concepts. *Proceedings of the Annual Meeting of the Cognitive Science Society*, 8.
 5. Brown, P. F., Cocke, J., Della Pietra, S. A., Della Pietra, V. J., Jelinek, F., Lafferty, J. D., Mercer, R. L., & Roossin, P. S. (1990). *A statistical approach to machine translation*.
 6. Chomsky, N. (1956). Three models for the description of language. *IRE Transactions on Information Theory*, 2(3), 113–124.
 7. Markov, A. A. (1913). An Example of Statistical Investigation of the Text 'Eugene Onegin' Concerning the Connection of Samples in Chains. *Izvestiya Imperatorskoi Akademii Nauk (Series VI)*, 7, 153–162.
 8. Bengio, Y., Ducharme, R., Vincent, P., & Jauvin, C. (2003). A neural probabilistic language model. *Journal of Machine Learning Research*, 3(Feb), 1137–1155.
 9. Mikolov, T., Yih, W.-tau, & Zweig, G. (2013). Linguistic regularities in continuous space word representations. *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, 746–751.
 10. Collobert, R., & Weston, J. (2008). A unified architecture for natural language processing: Deep neural networks with multitask learning. *Proceedings of the 25th International Conference on Machine Learning*, 160–167.
 11. Turian, J., Ratinov, L., & Bengio, Y. (2010). Word representations: a simple and general method for semi-supervised learning. *Proceedings of the 48th Annual Meeting of the Association for Computational Linguistics*, 384–394.
 12. Mikolov, T., Karafiát, M., Burget, L., Cernocký, J., & Khudanpur, S. (2010). Recurrent neural network based language model. *Interspeech*, 2(3), 1045–1048.
 13. Mnih, A., & Teh, Y. W. (2012). A fast and simple algorithm for training neural probabilistic language models. *ArXiv Preprint ArXiv:1206.6426*.

14. Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in Neural Information Processing Systems*, 25.
15. LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1989). Backpropagation applied to handwritten zip code recognition. *Neural Computation*, 1(4), 541–551.
16. Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., & Fei-Fei, L. (2009). Imagenet: A large-scale hierarchical image database. *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 248–255.
17. O'Hara, S., & Draper, B. A. (2011). Introduction to the bag of features paradigm for image classification and retrieval. *ArXiv Preprint ArXiv:1101.3354*.
18. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *ArXiv Preprint ArXiv:1301.3781*.
19. Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. *Advances in Neural Information Processing Systems*, 26.
20. Morin, F., & Bengio, Y. (2005). Hierarchical probabilistic neural network language model. *International Workshop on Artificial Intelligence and Statistics*, 246–252.
21. Gutmann, M., & Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, 297–304.
22. Mikolov, T. (2007). *Language modeling for speech recognition in czech* [PhD thesis]. Masters thesis, Brno University of Technology.
23. Mikolov, T., Kopecky, J., Burget, L., Glembek, O., & others. (2009). Neural network based language models for highly inflective languages. *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, 4725–4728.
24. Mnih, A., & Hinton, G. E. (2008). A scalable hierarchical distributed language model. *Advances in Neural Information Processing Systems*, 21.
25. Ritter, H., & Kohonen, T. (1989). Self-organizing semantic maps. *Biological Cybernetics*, 61(4), 241–254.
26. Kohonen, T. (1982). Self-organized formation of topologically correct feature maps. *Biological Cybernetics*, 43(1), 59–69.
27. Pennington, J., Socher, R., & Manning, C. D. (2014). Glove: Global vectors for word representation. *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, 1532–1543.
28. Kiros, R., Zhu, Y., Salakhutdinov, R. R., Zemel, R., Urtasun, R., Torralba, A., & Fidler, S. (2015). Skip-thought vectors. *Advances in Neural Information Processing Systems*, 28.
29. Lebre, R., & Collobert, R. (2013). Word emdeddings through hellinger PCA. *ArXiv Preprint ArXiv:1312.5542*.
30. Dhillon, P. S., Foster, D. P., & Ungar, L. H. (2015). Eigenwords: spectral word embeddings. *J. Mach. Learn. Res.*, 16, 3035–3078.
31. Melamud, O., Goldberger, J., & Dagan, I. (2016). context2vec: Learning generic context embedding with bidirectional LSTM. *Proceedings of the 20th SIGNLL Conference on Computational Natural Language Learning*, 51–61.

32. McCann, B., Bradbury, J., Xiong, C., & Socher, R. (2017). Learned in translation: Contextualized word vectors. *Advances in Neural Information Processing Systems*, 30.
33. Peters, M. E., Neumann, M., Zettlemoyer, L., & Yih, W.-tau. (2018). Dissecting contextual word embeddings: Architecture and representation. *ArXiv Preprint ArXiv:1808.08949*.
34. Waibel, A., Hanazawa, T., Hinton, G., Shikano, K., & Lang, K. J. (1989). Phoneme recognition using time-delay neural networks. In *IEEE transactions on acoustics, speech, and signal processing* (pp. 35–61). Psychology Press.
35. Bourlard, H., & Kamp, Y. (1988). Auto-association by multilayer perceptrons and singular value decomposition. *Biological Cybernetics*, 59(4), 291–294.
36. Elman, J. L. (1990). Finding structure in time. *Cognitive Science*, 14(2), 179–211.
37. Jordan, M. I. (1997). Serial order: A parallel distributed processing approach. In *Advances in psychology* (Vol. 121, pp. 471–495). Elsevier.
38. Hochreiter, S., & Schmidhuber, J. (1997). Long short-term memory. *Neural Computation*, 9(8), 1735–1780.
39. Sutskever, I., Martens, J., & Hinton, G. E. (2011). Generating text with recurrent neural networks. *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, 1017–1024.
40. Graves, A. (2013). Generating sequences with recurrent neural networks. *ArXiv Preprint ArXiv:1308.0850*.
41. Bengio, Y., Simard, P., & Frasconi, P. (1994). Learning long-term dependencies with gradient descent is difficult. *IEEE Transactions on Neural Networks*, 5(2), 157–166.
42. Kalchbrenner, N., & Blunsom, P. (2013). Recurrent continuous translation models. *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 1700–1709.
43. Kalchbrenner, N., & Blunsom, P. (2013). Recurrent convolutional neural networks for discourse compositionality. *ArXiv Preprint ArXiv:1306.3584*.
44. Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *ArXiv Preprint ArXiv:1406.1078*.
45. Sutskever, I., Vinyals, O., & Le, Q. V. (2014). Sequence to sequence learning with neural networks. *Advances in Neural Information Processing Systems*, 27.
46. Baum, L. E., & Petrie, T. (1966). Statistical inference for probabilistic functions of finite state Markov chains. *The Annals of Mathematical Statistics*, 37(6), 1554–1563.
47. Socher, R., Perelygin, A., Wu, J., Chuang, J., Manning, C. D., Ng, A. Y., & Potts, C. (2013). Recursive deep models for semantic compositionality over a sentiment treebank. *Proceedings of the 2013 Conference on Empirical Methods in Natural Language Processing*, 1631–1642.
48. Gammernan, A., Vovk, V., & Vapnik, V. (2013). Learning by transduction. *ArXiv Preprint ArXiv:1301.7375*.
49. Cho, K., Van Merriënboer, B., Bahdanau, D., & Bengio, Y. (2014). On the properties of neural machine translation: Encoder-decoder approaches. *ArXiv Preprint ArXiv:1409.1259*.

50. Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *ArXiv Preprint ArXiv:1409.0473*.
51. Larochelle, H., & Hinton, G. E. (2010). Learning to combine foveal glimpses with a third-order Boltzmann machine. *Advances in Neural Information Processing Systems*, 23.
52. Denil, M., Bazzani, L., Larochelle, H., & de Freitas, N. (2012). Learning where to attend with deep architectures for image tracking. *Neural Computation*, 24(8), 2151–2184.
53. Mnih, V., Heess, N., Graves, A., & Kavukcuoglu, K. (2014). Recurrent models of visual attention. *Advances in Neural Information Processing Systems*, 27.
54. Luong, M.-T., Pham, H., & Manning, C. D. (2015). Effective approaches to attention-based neural machine translation. *ArXiv Preprint ArXiv:1508.04025*.
55. Cheng, J., Dong, L., & Lapata, M. (2016). Long short-term memory-networks for machine reading. *ArXiv Preprint ArXiv:1601.06733*.
56. Zaremba, W., & Sutskever, I. (2014). Learning to execute. *ArXiv Preprint ArXiv:1410.4615*.
57. Lin, Z., Feng, M., Santos, C. N. dos, Yu, M., Xiang, B., Zhou, B., & Bengio, Y. (2017). A structured self-attentive sentence embedding. *ArXiv Preprint ArXiv:1703.03130*.
58. Seo, M., Kembhavi, A., Farhadi, A., & Hajishirzi, H. (2016). Bidirectional attention flow for machine comprehension. *ArXiv Preprint ArXiv:1611.01603*.
59. Britz, D., Goldie, A., Luong, M.-T., & Le, Q. (2017). Massive exploration of neural machine translation architectures. *ArXiv Preprint ArXiv:1703.03906*.
60. Jozefowicz, R., Vinyals, O., Schuster, M., Shazeer, N., & Wu, Y. (2016). Exploring the limits of language modeling. *ArXiv Preprint ArXiv:1602.02410*.
61. Ba, J. L., Kiros, J. R., & Hinton, G. E. (2016). Layer normalization. *ArXiv Preprint ArXiv:1607.06450*.
62. He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 770–778.
63. Gehring, J., Auli, M., Grangier, D., Yarats, D., & Dauphin, Y. N. (2017). Convolutional sequence to sequence learning. *International Conference on Machine Learning*, 1243–1252.
64. Radford, A., Narasimhan, K., Salimans, T., Sutskever, I., & others. (2018). *Improving language understanding by generative pre-training*.
65. Ng, A., & Jordan, M. (2001). On discriminative vs. generative classifiers: A comparison of logistic regression and naive bayes. *Advances in Neural Information Processing Systems*, 14.
66. Devlin, J., Chang, M.-W., Lee, K., & Toutanova, K. (2019). Bert: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, 4171–4186.
67. Taylor, W. L. (1953). Cloze procedure: A new tool for measuring readability. *Journalism Quarterly*, 30(4), 415–433.

68. Reddy, S., Chen, D., & Manning, C. D. (2019). Coqa: A conversational question answering challenge. *Transactions of the Association for Computational Linguistics*, 7, 249–266.
69. Nallapati, R., Zhou, B., Gulcehre, C., Xiang, B., & others. (2016). Abstractive text summarization using sequence-to-sequence rnns and beyond. *ArXiv Preprint ArXiv:1602.06023*.
70. Zellers, R., Holtzman, A., Bisk, Y., Farhadi, A., & Choi, Y. (2019). Hellaswag: Can a machine really finish your sentence? *ArXiv Preprint ArXiv:1905.07830*.
71. Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. D. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., & others. (2021). Evaluating large language models trained on code. *ArXiv Preprint ArXiv:2107.03374*.
72. Paperno, D., Kruszewski, G., Lazaridou, A., Pham, Q. N., Bernardi, R., Pezzelle, S., Baroni, M., Boleda, G., & Fernández, R. (2016). The LAMBADA dataset: Word prediction requiring a broad discourse context. *ArXiv Preprint ArXiv:1606.06031*.
73. Cobbe, K., Kosaraju, V., Bavarian, M., Chen, M., Jun, H., Kaiser, L., Plappert, M., Tworek, J., Hilton, J., Nakano, R., & others. (2021). Training verifiers to solve math word problems. *ArXiv Preprint ArXiv:2110.14168*.
74. Ramachandran, P., Liu, P. J., & Le, Q. V. (2016). Unsupervised pretraining for sequence to sequence learning. *ArXiv Preprint ArXiv:1611.02683*.
75. Hinton, G., Deng, L., Yu, D., Dahl, G. E., Mohamed, A.-rahman, Jaitly, N., Senior, A., Vanhoucke, V., Nguyen, P., Sainath, T. N., & others. (2012). Deep neural networks for acoustic modeling in speech recognition. *IEEE Signal Processing Magazine*, 29(6), 82–97.
76. Bengio, Y., Lamblin, P., Popovici, D., & Larochelle, H. (2006). Greedy layer-wise training of deep networks. *Advances in Neural Information Processing Systems*, 19.
77. Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., Sutskever, I., & others. (2019). Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8), 9.
78. Brown, T., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., & others. (2020). Language models are few-shot learners. *Advances in Neural Information Processing Systems*, 33, 1877–1901.
79. Palatucci, M., Pomerleau, D., Hinton, G. E., & Mitchell, T. M. (2009). Zero-shot learning with semantic output codes. *Advances in Neural Information Processing Systems*, 22.
80. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F. L., Almeida, D., Altschmidt, J., Altman, S., Anadkat, S., & others. (2023). Gpt-4 technical report. *ArXiv Preprint ArXiv:2303.08774*.
81. Hendrycks, D., Burns, C., Basart, S., Critch, A., Li, J., Song, D., & Steinhardt, J. (2020). Aligning ai with shared human values. *ArXiv Preprint ArXiv:2008.02275*.
82. Gehman, S., Gururangan, S., Sap, M., Choi, Y., & Smith, N. A. (2020). Realtoxicityprompts: Evaluating neural toxic degeneration in language models. *ArXiv Preprint ArXiv:2009.11462*.
83. Christiano, P. F., Leike, J., Brown, T., Martic, M., Legg, S., & Amodei, D. (2017). Deep reinforcement learning from human preferences. *Advances in Neural Information Processing Systems*, 30.

84. Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., Van Den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., & others. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), 484–489.
85. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *ArXiv Preprint ArXiv:1312.5602*.
86. Berner, C., Brockman, G., Chan, B., Cheung, V., Debiak, P., Dennison, C., Farhi, D., Fischer, Q., Hashme, S., Hesse, C., & others. (2019). Dota 2 with large scale deep reinforcement learning. *ArXiv Preprint ArXiv:1912.06680*.
87. Stiennon, N., Ouyang, L., Wu, J., Ziegler, D., Lowe, R., Voss, C., Radford, A., Amodei, D., & Christiano, P. F. (2020). Learning to summarize with human feedback. *Advances in Neural Information Processing Systems*, 33, 3008–3021.
88. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., & Klimov, O. (2017). Proximal policy optimization algorithms. *ArXiv Preprint ArXiv:1707.06347*.
89. Ziegler, D. M., Stiennon, N., Wu, J., Brown, T. B., Radford, A., Amodei, D., Christiano, P., & Irving, G. (2019). Fine-tuning language models from human preferences. *ArXiv Preprint ArXiv:1909.08593*.
90. Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., & others. (2022). Training language models to follow instructions with human feedback. *Advances in Neural Information Processing Systems*, 35, 27730–27744.
91. Askell, A., Bai, Y., Chen, A., Drain, D., Ganguli, D., Henighan, T., Jones, A., Joseph, N., Mann, B., DasSarma, N., & others. (2021). A general language assistant as a laboratory for alignment. *ArXiv Preprint ArXiv:2112.00861*.
92. Team, G., Anil, R., Borgeaud, S., Alayrac, J.-B., Yu, J., Soricut, R., Schalkwyk, J., Dai, A. M., Hauth, A., Millican, K., & others. (2023). Gemini: a family of highly capable multimodal models. *ArXiv Preprint ArXiv:2312.11805*.
93. Chowdhery, A., Narang, S., Devlin, J., Bosma, M., Mishra, G., Roberts, A., Barham, P., Chung, H. W., Sutton, C., Gehrmann, S., & others. (2023). Palm: Scaling language modeling with pathways. *Journal of Machine Learning Research*, 24(240), 1–113.
94. Thoppilan, R., De Freitas, D., Hall, J., Shazeer, N., Kulshreshtha, A., Cheng, H.-T., Jin, A., Bos, T., Baker, L., Du, Y., & others. (2022). Lamda: Language models for dialog applications. *ArXiv Preprint ArXiv:2201.08239*.
95. Rae, J. W., Borgeaud, S., Cai, T., Millican, K., Hoffmann, J., Song, F., Aslanides, J., Henderson, S., Ring, R., Young, S., & others. (2021). Scaling language models: Methods, analysis & insights from training gopher. *ArXiv Preprint ArXiv:2112.11446*.
96. Touvron, H., Lavril, T., Izacard, G., Martinet, X., Lachaux, M.-A., Lacroix, T., Rozière, B., Goyal, N., Hambro, E., Azhar, F., & others. (2023). Llama: Open and efficient foundation language models. *ArXiv Preprint ArXiv:2302.13971*.
97. Touvron, H., Martin, L., Stone, K., Albert, P., Almahairi, A., Babaei, Y., Bashlykov, N., Batra, S., Bhargava, P., Bhosale, S., & others. (2023). Llama 2: Open foundation and fine-tuned chat models. *ArXiv Preprint ArXiv:2307.09288*.

98. Guo, D., Yang, D., Zhang, H., Song, J., Zhang, R., Xu, R., Zhu, Q., Ma, S., Wang, P., Bi, X., & others. (2025). Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *ArXiv Preprint ArXiv:2501.12948*.
99. Ling, W., Yogatama, D., Dyer, C., & Blunsom, P. (2017). Program induction by rationale generation: Learning to solve and explain algebraic word problems. *ArXiv Preprint ArXiv:1705.04146*.
100. Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., & others. (2022). Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35, 24824–24837.
101. Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., & Amodei, D. (2020). Scaling laws for neural language models. *ArXiv Preprint ArXiv:2001.08361*.
102. Wei, J., Tay, Y., Bommasani, R., Raffel, C., Zoph, B., Borgeaud, S., Yogatama, D., Bosma, M., Zhou, D., Metzler, D., & others. (2022). Emergent abilities of large language models. *ArXiv Preprint ArXiv:2206.07682*.
103. Hendrycks, D., Burns, C., Kadavath, S., Arora, A., Basart, S., Tang, E., Song, D., & Steinhardt, J. (2021). Measuring mathematical problem solving with the math dataset. *ArXiv Preprint ArXiv:2103.03874*.
104. Jumper, J., Evans, R., Pritzel, A., Green, T., Figurnov, M., Ronneberger, O., Tunyasuvunakool, K., Bates, R., Žídek Augustin, Potapenko, A., & others. (2021). Highly accurate protein structure prediction with AlphaFold. *Nature*, 596(7873), 583–589.